(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 September 2002 (19.09.2002)

PCT

(10) International Publication Number
WO 02/073937 A2

(51) International Patent Classification⁷: H04M

(21) International Application Number: PCT/US02/08106

(22) International Filing Date: 14 March 2002 (14.03.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/275,846	14 March 2001 (14.03.2001)	US
60/289,600	7 May 2001 (07.05.2001)	US
60/295,060	1 June 2001 (01.06.2001)	US

(71) Applicant: MERCURY COMPUTER SYSTEMS, INC. [US/US]; 199 Riverneck Road, Chelmsford, MA 01824 (US).

(72) Inventor: OATES, John, H.; 59B Seaverns Bridge Road, Amherst, NH 03031 (US).

(74) Agents: POWSNER, David, J. et al.; Nutter, McClennen & Fish LLP, One International Place, Boston, MA 02110-2699 (US).

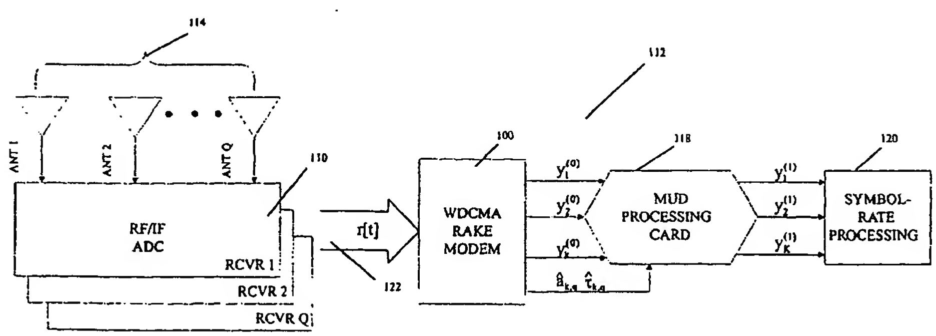
(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: WIRELESS COMMUNICATIONS METHODS AND SYSTEMS FOR LONG-CODE AND OTHER SPREAD SPECTRUM WAVEFORM PROCESSING



(57) Abstract: The invention provides improved CDMA, WCDMA (UTMS) or other spread spectrum communication systems of the type that processes one or more spread-spectrum waveforms, each representative of a waveform received from a respective user (or other transmitting device). The improvement is characterized by a first logic element that generates a residual composite spread-spectrum waveform as a function of an arithmetic difference between a composite spread-spectrum waveform for all users (or other transmitters) and an estimated spread-spectrum waveform for each user. It is further characterized by one or more second logic elements that generate, for at least a selected user (or other transmitter), a refined spread-spectrum waveform as a function of a sum of the residual composite spread-spectrum waveform and the estimated spread-spectrum waveform for that user.

WO 02/073937 A2

BEST AVAILABLE COPY

Wireless Commmunications Methods and Systems for Long-Code and Other Spread Spectrum Waveform Processing

Background of the Invention

This application claims the benefit of priority of (i) US Provisional Application Serial No. 60/275,846 filed March 14, 2001, entitled "Improved Wireless Communications Systems and Methods"; (ii) US Provisional Application Serial No. 60/289,600 filed May 7, 2001, entitled "Improved Wireless Communications Systems and Methods Using Long-Code Multi-User Detection"' and (iii) US Provisional Application Serial Number. 60/295,060 filed June 1, 2001 entitled "Improved Wireless Communications Systems and Methods for a Communications Computer," the teachings all of which are incorporated herein by reference.

The invention pertains to wireless communications and, more particularly, by way of example, to methods and apparatus providing multiple user detection for use in code division multiple access (CDMA) communications. The invention has application, by way of non-limiting example, in improving the capacity of cellular phone base stations.

Code-division multiple access (CDMA) is used increasingly in wireless communications. It is a form of multiplexing communications, e.g., between cellular phones and base stations, based on distinct digital codes in the communication signals. This can be contrasted with other wireless protocols, such as frequency-division multiple access and time-division multiple access, in which multiplexing is based on the use of orthogonal frequency bands and orthogonal time-slots, respectively.

A limiting factor in CDMA communication and, in particular, in so-called direct sequence CDMA (DS-CDMA) communication, is the interference between multiple cellular phone users in the same geographic area using their phones at the same time, which is referred to as multiple access interference (MAI). Multiple access interference has an effect of limiting the capacity of cellular phone base stations, driving service quality below acceptable levels when there are too many users.

A technique known as multi-user detection (MUD) is intended to reduce multiple access interference and, as a consequence, increases base station capacity. It can reduce interference not only between multiple transmissions of like strength, but also that caused by users so close to the base station as to otherwise overpower signals from other users (the so-called near/far problem). MUD generally functions on the principle that signals from multiple simultaneous users can be jointly used to improve detection of the signal from any single user. Many

forms of MUD are discussed in the literature; surveys are provided in Moshavi, "Multi-User Detection for DS-CDMA Systems," IEEE Communications Magazine (October, 1996) and Duel-Hallen et al, "Multiuser Detection for CDMA Systems," IEEE Personal Communications (April 1995). Though a promising solution to increasing the capacity of cellular phone base stations, MUD techniques are typically so computationally intensive as to limit practical application.

An object of this invention is to provide improved methods and apparatus for wireless communications. A related object is to provide such methods and apparatus for multi-user detection or interference cancellation in code-division multiple access communications.

A further related object is to provide such methods and apparatus as provide improved short-code and/or long-code CDMA communications.

A further object of the invention is to provide such methods and apparatus as can be cost-effectively implemented and as require minimal changes in existing wireless communications infrastructure.

A still further object of the invention is to provide methods and apparatus for executing multi-user detection and related algorithms in real-time.

A still further object of the invention is to provide such methods and apparatus as manage faults for high-availability.

Summary of the Invention

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Implicit Waveform Subtraction

The foregoing and other objects are among those attained by the invention which provides, in one aspect, an improved spread-spectrum communication system of the type that processes one or more spread-spectrum waveforms, e.g., a CDMA transmissions, each representative of a waveform received from, or otherwise associated with, a respective user (or other transmitting device). The improvement is characterized by a first logic element, e.g., operating in conjunction with a wireless base station receiver and/or modem, that generates a residual composite spread-spectrum waveform as a function of a composite spread-spectrum waveform and an estimated composite spread-spectrum waveform. It is further characterized by one or more second logic elements that generate, for at least a selected user (or other transmitter), a refined matched-filter detection statistic as a function of the residual composite spread-spectrum waveform generated by the first logic element and a characteristic of an estimate of the selected user's spread-spectrum waveform.

Related aspects of the invention as described above provide a system as described above in which the first logic element comprises arithmetic logic that generates the composite spread-spectrum waveform based on a relation

$$r_{res}^{(n)}[t] \equiv r[t] - \hat{r}^{(n)}[t]$$

wherein

$r_{res}^{(n)}[t]$ is the residual composite spread-spectrum waveform,

$r[t]$ represents the composite spread-spectrum waveform,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

t is a sample time period, and

n is an iteration count,

The estimated composite spread-spectrum waveform, according to further related aspects, can be pulse-shaped and based on estimated complex amplitudes, estimated symbols, and codes encoded within the user waveforms.

Still further aspects of the invention provide improved spread-spectrum communication systems as described above in which the one or more second logic elements comprise rake logic and summation logic, which generate the refined matched-filter detection statistic for at least the selected user based on a relation

$$y_k^{(n+1)}[m] = A_k^{(n)^2} \cdot \hat{b}_k^{(n)}[m] + y_{res,k}^{(n)}[m]$$

wherein

$A_k^{(n)^2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period ,

$y_{res,k}^{(n)}[m]$ represents a residual matched-filter detection statistic for the k^{th} user, and

n is an iteration count.

Further related aspects of the invention provide improved systems as described above wherein the refined matched-filter detection statistics for each user is iteratively generated. Related aspects of the invention provide such systems in which the user spread-spectrum waveform for at least a selected user is generated by a receiver that operates on long-code CDMA signals.

Further aspects of the invention provide a spread spectrum communication system, e.g., of the type described above, having a first logic element which generates an estimated composite spread-spectrum waveform as a function of estimated user complex channel amplitudes, time lags, and user codes. A second logic element generates a residual composite spread-spectrum waveform a function of a composite user spread-spectrum waveform and the estimated composite spread-spectrum waveform. One or more third logic elements generate a refined matched-filter detection statistic for at least a selected user as a function of the residual composite spread-spectrum waveform and a characteristic of an estimate of the selected user's spread-spectrum waveform.

A related aspects of the invention provides such systems in which the first logic element generates the estimated re-spread waveform based on a relation

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v} \sum_{p=1}^{L} \sum_{r} \delta[t - \hat{\tau}_{kp}^{(n)} - rN_c] \cdot \hat{a}_{kp}^{(n)} \cdot c_k[r] \cdot \hat{b}_k^{(n)}[\lfloor r / N_k \rfloor]$$

wherein

K_v is a number of simultaneous dedicated physical channels for all users,

$\delta[t]$ is a discrete-time delta function,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

$c_k[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user ,

N_k is a spreading factor for the k^{th} user,

t is a sample time index,

L is a number of multi-path components.,

N_c is a number of samples per chip, and

n is an iteration count.

Related aspects of the invention provide systems as described above wherein the first logic element comprises arithmetic logic that generates the estimated composite spread-spectrum waveform based on the relation

$$\hat{r}^{(n)}[t] = \sum_{r} g[r]\rho^{(n)}[t-r],$$

wherein

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

$g[t]$ represents a raised-cosine pulse shape.

Related aspects of the invention provide such systems that comprise a CDMA base station, e.g., of the type for use in relaying voice and data traffic from cellular phone and/or modem users. Still further aspects of the invention provide improved spread spectrum communication systems as described above in which the user waveforms are encoded using long-code CDMA protocols.

Still other aspects of the invention provide methods multiple user detection in a spread-spectrum communication system paralleling the operations described above.

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Matched-filter Outputs

Further aspects of the invention provide an improved spread spectrum communication system, e.g., of the type described above, having first logic element operating in conjunction with a wireless base station receiver and/or modem, that generates an estimated composite spread-spectrum waveform as a function of user waveform characteristics, e.g., estimated complex amplitudes, time lags, symbols and code. The invention is further characterized by one or more second logic elements that generate for at least a selected user a refined matched-filter detection statistic as a function of a difference between a first matched-filter detection statistic for that user and an estimated matched-filter detection statistic—the latter of which is a function of the estimated composite spread-spectrum waveform generated by the first logic element.

Related aspects of the invention as described above provide for improved wireless communications wherein each of the second logic elements generate the refined matched-filter detection statistic for the selected user as a function of a difference between (i) a sum of the first matched-filter detection statistic for that user and a characteristic of an estimate of that user's

spread-spectrum waveform, and (ii) the estimated matched-filter detection statistic for that user based on the estimated composite spread-spectrum waveform.

Further related aspects of the invention provide systems as described above in which the second logic elements comprise rake logic and summation logic which generates refined matched-filter detection statistics for at least a selected user in accord with the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_k^{(n)}[m] - y_{est,k}^{(n)}[m]$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the mth symbol period,

$y_k^{(n)}[m]$ represents the first matched-filter detection statistic,

$y_{est,k}^{(n)}[m]$ represents the estimated matched-filter detection statistic, and

n is an iteration count.

Other related aspects of the invention include generating the refined matched-filter detection statistic for the selected user and iteratively refining that detection statistic zero or more times.

Related aspects of the invention as described above provide for improved wireless communications methods wherein an estimated composite spread-spectrum waveform is based on the relation

$$y_{est,k}^{(n)}[m] \equiv \mathrm{Re}\left\{\sum_{p=1}^{L} \hat{a}_{kp}^{(n)H} \cdot \frac{1}{2N_k} \sum_{r=0}^{N_k-1} \hat{r}^{(n)}[rN_c + \hat{\tau}_{kp}^{(n)} + mT_k] \cdot c_{km}^{*}[r]\right\},$$

wherein

L is a number of multi-path components,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

N_k is a spreading factor for the k^{th} user,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

N_c is a number of samples per chip,

$\hat{\tau}^{(n)}_{kp}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user,

m is a symbol period,

T_k is a data bit duration,

n is an iteration count, and

$c_{km}[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels.

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Pre-maximal Combination Matched Filter Outputs

Still further aspects of the invention provide improved-spread spectrum communication systems, e.g., of the type described above, having one or more first logic elements, e.g., operating in conjunction with a wireless base station receiver and/or modem, that generate a first complex channel amplitude estimate corresponding to at least a selected user and a selected finger of a rake receiver that receives the selected user waveforms. One or more second logic elements generate an estimated composite spread-spectrum waveform that is a function of one or more complex channel amplitudes, estimated delay lags, estimated symbols, and/or codes of the one or more user spread-spectrum waveforms. One or more third logic elements generate a second pre-combination matched-filter detection statistic for at least a selected user and for at least a selected finger as a function of a first pre-combination matched-filter detection statistic for that user and a pre-combination estimated matched-filter detection statistic for that user.

Related aspects of the invention provide systems as described above in which one or more fourth logic elements generate a second complex channel amplitude estimate corresponding to at least a selected user and at least selected finger.

Still further aspects of the invention provide systems as described above in which the third logic elements generate the second pre-combination matched-filter detection statistic for at least the selected user and at least the selected finger as a function of a difference between (i) the sum of the first pre-combination matched-filter detection statistic for that user and that finger and a characteristic of an estimate of the selected user's spread-spectrum waveform and (ii) the pre-combination estimated matched-filter detection statistic for that user and that finger.

Related aspects of the invention as described above provide for the first logic elements generating a complex channel amplitude estimated corresponding to at least a selected user and at least a selected finger of a rake receiver that receives the selected user waveforms based on a relation

$$\hat{a}_{kp}^{(n)} \equiv \sum_{s} w[s] \cdot \frac{1}{N_p} \sum_{m=0}^{N_p-1} y_{kp}^{(n)}[m+Ms] \cdot \hat{b}_k^{(n)}[m+Ms]$$

wherein

$\hat{a}_{kp}^{(n)}$ is a complex channel amplitude estimate corresponding to the p^{th} finger of the k^{th} user,

$w[s]$ is a filter,

N_p is a number of symbols,

$y_{kp}^{(n)}[m]$ is a first pre-combination matched-filter detection statistic corresponding to the p^{th} finger of the k^{th} user for the m^{th} symbol period,

M is a number of symbols per slot,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

m is a number symbol period index,

s is a slot index, and

n is an iteration count.

Further related aspects of the invention as described above provide for one or more second logic elements, each coupled with a first logic element and using the complex channel amplitudes generated therefrom to generate an estimated composite re-spread waveform based on the relation

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v} \sum_{p=1}^{L} \sum_{r} \delta[t - \hat{\tau}_{kp}^{(n)} - rN_c] \cdot \hat{a}_{kp}^{(n)} \cdot c_k[r] \cdot \hat{b}_k^{(n)}[\lfloor r / N_k \rfloor],$$

wherein

K_v is a number of simultaneous dedicated physical channels for all users,

$\delta[t]$ is a discrete-time delta function,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

$c_k[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user ,

N_k is a spreading factor for the k^{th} user,

t is a sample time index,

L is a number of multi-path components.,

N_c is a number of samples per chip, and

n is an iteration count.

Further related aspects of the invention provide systems as described above in which the second logic element comprises arithmetic logic that generates the estimated composite spread-spectrum waveform based on a relation

$$\hat{r}^{(n)}[t] = \sum_{r} g[r]\rho^{(n)}[t-r]$$

wherein

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

$g[t]$ represents a pulse shape.

Still further related aspects of the invention provide systems as described above in which the third logic elements comprise arithmetic logic that generates the second pre-combination matched-filter detection statistic based on the relation

$$y_{kp}^{(n+1)}[m] \equiv \hat{a}_{kp}^{(n)} \cdot \hat{b}_{k}^{(n)}[m] + y_{kp}^{(n)}[m] - y_{est,kp}^{(n)}[m]$$

wherein

$y_{kp}^{(n+1)}[m]$ represents the pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$\hat{a}_{kp}^{(n)}$ is the complex channel amplitude for the p^{th} finger for the k^{th} user,

$\hat{b}_{k}^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{kp}^{(n)}[m]$ represents the first pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$y_{est,kp}^{(n)}[m]$ represents the pre-combination estimated matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period, and

n is an iteration count.

Still further aspects of the invention provide methods of operating multiuser detector logic, wireless base stations and/or other wireless receiving devices or systems operating in the manner of the apparatus above. Further aspects of the invention provide such systems in which the first and second logic elements are implemented on any of processors, field programmable

gate arrays, array processors and co-processors, or any combination thereof. Other aspects of the invention provide for interatively refining the pre-combination matched-filter detection statistics zero or more time.

Other aspects of the invention provide methods for an improved spread-spectrum communication system as the type described above.

Brief Description of the Illustrated Embodiment

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1 is a block diagram of components of a wireless base-station utilizing a multi-user detection apparatus according to the invention.

Figure 2 is a detailed diagram of a modem of the type that receives spread-spectrum waveforms and generates a baseband spectrum waveform together with amplitude and time lag estimates as used by the invention.

Figures 3 and 4 depict methods according to the invention for multiple user detection using explicitly regenerated user waveforms which are added to a residual waveform.

Figure 5 depicts methods according to the invention for multiple user detection in which user waveforms are regenerated from a composite spread-spectrum pulsed-shaped waveform.

Figure 6 depicts methods according to the invention for multiple user detection using matched-filter outputs where a composite spread-spectrum pulse-shaped waveform is rake-processed.

Figure 7 depicts methods according to the invention for multiple user detection using pre-maximum ratio combined matched-filter output, where a composite spread-spectrum pulse-shaped waveform is rake processed.

Figure 8 depicts an approach for processing user waveforms using full or partial decoding at various time-transmission intervals based on user class.

Figure 9 depicts an approach for combining multi-path data across received frame boundaries to preserve the number of multi user detection processing frame counts.

Figure 10 illustrates the mapping of rake receiver output to virtual to preserve spreading factor and number of data channels across multiple user detection processing frames where the data is linear and contiguous in memory.

Figure 11 depicts a long-code loading implementation utilizing pipelined processing and a triple-iteration of refinement in a system according to the invention; and

Figure 12 illustrates skewing of multiple user waveforms.

Detailed Description of the Illustrated Embodiment

Code-division multiple access (CDMA) waveforms or signals transmitted, e.g., from a user cellular phone, modem or other CDMA signal source, can become distorted by, and undergo amplitude fades and phase shifts due to phenomena such as scattering, diffraction and/or reflection off buildings and other natural and man-made structures. This includes CDMA, DS/CDMA, IS-95 CDMA, CDMAOne, CDMA2000 1X, CDMA2000 1xEV-DO, WCDMA (or UTMS), and other forms of CDMA, which are collectively referred to hereinafter as CDMA or WCDMA. Often the user or other source (collectively, "user") is also moving, e.g., in a car or train, adding to the resulting signal distortion by alternately increasing and decreasing the distances to and numbers of building, structures and other distorting factors between the user and the base station.

In general, because each user signal can be distorted several different ways en route to the base station or other receiver (hereinafter, collectively, "base station"), the signal may be received in several components, each with a different time lag or phase shift. To maximize detection of a given user signal across multiple tag lags, a rake receiver is utilized. Such a receiver is coupled to one or more RF antennas (which serve as a collection point(s) for the time-lagged components) and includes multiple fingers, each designed to detect a different multipath component of the user signal. By combining the components, e.g., in power or amplitude, the receiver permits the original waveform to be discerned more readily, e.g., by downstream elements in the base station and/or communications path.

A base station must typically handle multiple user signals, and detect and differentiate among signals received from multiple simultaneous users, e.g., multiple cell phone users in the vicinity of the base station. Detection is typically accomplished through use of multiple rake receivers, one dedicated to each user. This strategy is referred to as single user detection (SUD). Alternately, one larger receiver can be assigned to demodulate the totality of users jointly. This strategy is referred to as multiple user detection (MUD). Multiple user detection can be accomplished through various techniques which aim to discern the individual user signals and to reduce signal outage probability or bit-error rates (BER) to acceptable levels.

However, the process has heretofore been limited due to computational complexities which can increase exponentially with respect to the number of simultaneous users. Described below are embodiments that overcome this, providing, for example, methods for multiple user detection wherein the computational complexity is linear with respect to the number of users and providing, by way of further example, apparatus for implementing those and other methods that improve the throughput of CDMA and other spread-spectrum receivers. The illus-

trated embodiments are implemented in connection with long-code CDMA transmitting and receiver apparatus; however those skilled in the art will appreciate that the methods and apparatus therein may be used in connection with short-code and other CDMA signalling protocols and receiving apparatus, as well as with other spread spectrum signalling protocols and receiving apparatus. In these regards and as used herein, the terms long-code and short-code are used in their conventional sense: the former referring to codes that exceed one symbol period; the latter, to codes that are a single symbol period or less.

Five embodiments of long-code regeneration and waveform refinement are presented herein. The first two may be referred to as a base-line embodiment and a residual signal embodiment. The remaining three embodiments use implicit waveform subtraction, matched-filter outputs rather than antenna streams and pre-maximum ratio combination of matched-filter outputs. It will be appreciated by those skilled in the art, that other modifications to these techniques can be implemented that produce the like results based on modifications of the methods described herein.

Figure 1 depicts components of a wireless base station 100 of the type in which the invention is practiced. The base station 100 includes an antenna array 114, radio frequency/ intermediate frequency (RF/IF) analog-to-digital converter (ADC), multi-antenna receivers 110, rake modems 112, MUD processing logic 118 and symbol rate processing logic 120, coupled as shown.

Antenna array 114 and receivers 110 are conventional such devices of the type used in wireless base stations to receive wideband CDMA (hereinafter "WCDMA") transmissions from multiple simultaneous users (here, identified by numbers 1 through K). Each RF/IF receiver (e.g., 110) is coupled to antenna or antennas 114 in the conventional manner known in the art, with one RF/IF receiver 110 allocated for each antenna 114. Moreover, the antennas are arranged per convention to receive components of the respective user waveforms along different lagged signal paths discussed above. Though only three antennas 114 and three receivers 110 are shown, the methods and systems taught herein may be used with any number of such devices, regardless of whether configured as a base station, a mobile unit or otherwise. Moreover, as noted above, they may be applied in processing other CDMA and wireless communications signals.

Each RF/IF receiver 110 routes digital data to each modem 112. Because there are multiple antennas, here, Q of them, there are typically Q separate channel signals communicated to each modem card 112.

Generally, each user generating a WCDMA signal (or other subject wireless communication signal) received and processed by the base station is assigned a unique long-code code sequence for purpose of differentiating between the multiple user waveforms received at the basestation, and each user is assigned a unique rake modem 112 for purpose of demodulating the user's received signal. Each modem 112 may be independent, or may share resources from a pool. The rake modems 112 process the received signal components along fingers, with each receiver discerning the signals associated with that receiver's respective user codes. The received signal components are denoted here as $r_{kq}[t]$ denoting the channel signal (or waveform) from the k^{th} user from the q^{th} antenna, or $r_k[t]$ denoting all channel signals (or waveforms) originating from the k^{th} user, in which case $r_k[t]$ is understood to be a column vector with one element for each of the Q antennas. The modems 112 process the received signals $r_k[t]$ to generate detection statistics $y_k^{(0)}[m]$ for the k^{th} user for the mth symbol period. To this end, the modems 122 can, for example, combine the components $r_{kq}[t]$ by power, amplitude or otherwise, in the conventional manner to generate the respective detection statistics $y_k^{(0)}[m]$. In the course of such processing, each modem 112 determines the amplitude (denoted herein as a) of and time lag (denoted herein as τ) between the multiple components of the respective user channel. The modems 112 can be constructed and operated in the conventional manner known in the art, optionally, as modified in accord with the teachings of some of the embodiments below.

The modems 112 route their respective user detection statistics $y_k^{(0)}[m]$, as well as the amplitudes and time lags, to common user detection (MUD) 118 logic constructed and operated as described in the sections that follow. The MUD logic 118 processes the received signals from each modem 112 to generate a refined output, $y_k^{(1)}[m]$, or more generally, $y_k^{(n)}[m]$, where n is an index reflecting the number of times the detection statistics are iteratively or regeneratively processed by the logic 118. Thus, whereas the detection statistic produced by the modems is denoted as $y_k^{(0)}[m]$ indicating that there has been no refinement, those generated by processing the $y_k^{(0)}[m]$ detection statistics with logic 118 are denoted $y_k^{(1)}[m]$, those generated by processing the $y_k^{(1)}[m]$ detection statistics with logic 118 are denoted $y_k^{(2)}[m]$, and so forth. Further waveforms used and generated by logic 118 are similarly denoted, e.g., $r^{(n)}[t]$.

Though discussed below are embodiments in which the logic 118 is utilized only once, i.e., to generate $y_k^{(1)}[m]$ from $y_k^{(0)}[m]$, other embodiments may employ that logic 118 multiple times to generate still more refined detection statistics, e.g., for wireless communications applications requiring lower bit error rates (BER). For example, in some implementations, a single logic stage 118 is used for voice applications, whereas two or more logic stages are used for data applications. Where multiple stages are employed, each may be carried out using the

same hardware device (e.g., processor, co-processor or field programmable gate array) or with a successive series of such devices.

The refined user detection statistics, e.g., $y_k^{(1)}[m]$ or more generally $y_k^{(n)}[m]$, are communicated by the MUD process 118 to a symbol process 120. This determines the digital information contained within the detection statistics, and processes (or otherwise directs) that information according to the type of user class for which the user belongs, e.g., voice or data user, all in the conventional manner.

Though the discussion herein focuses on use of MUD logic 118 in a wireless base station, those skilled in the art will appreciate that the teachings hereof are equally applicable to MUD detection in any other CDMA signal processing environment such as, by way of non-limiting example, cellular phones and modems. For convenience, such cellular base stations other environments are referred to herein as "base stations."

Referring to Figure 2, modem 112 receives the channel-signals $r[t]$ 112 from the RF/IC receiver (Figure 1). The signals are first input into a searcher receiver 212. The searcher receiver analyzes the digital waveform input, and estimates a time offset $\hat{\tau}_{kp}^{(n)}$ for each signal component (e.g. for each finger). As those skilled in the art will appreciate, the "hat" or ^ symbol denotes estimated values. The time offset for each antenna channel is communicated to a corresponding rake receiver 214.

The rake receiver receivers 214 receive both the digital signals $r[t]$ from the RF/IF receivers, and the time offsets, $\hat{\tau}_{kp}^{(n)}$. The receivers 214 calculate the pre-combination matched-filter detection statistics, $y_{kp}^{(0)}[m]$, and estimate signal amplitude, $\hat{a}_{kp}^{(n)}$, for each of the signals. The amplitudes are complex in value, and hence include both the magnitude and phase information. The pre-combination matched-filter detection statistics, $y_{kp}^{(0)}[m]$, and the amplitudes $\hat{a}_{kp}^{(n)}$ for each finger receiver 212, are routed to a maximal ratio combining (MRC) 216 process and combined to form a first approximation of the symbols transmitted by each user, denoted $y_k^{(0)}[m]$. While the MRC 216 process is utilized in the illustrated embodiment, other methods for combining the multiple signals are known in the art, e.g., optimal combining, equal gain combining and selection combining, among others, and can be used to achieve the same results.

At this point, it can be appreciated by one skilled in the art that each detection statistic, $y_k^{(0)}[m]$, contains not only the signal originating from user k, but also has components (e.g., interference and noise) that have originated in the channel (e.g., the environment in which the signal was propagated and/or in the receiving apparatus itself). Hence, it is further necessary

to differentiate each user's signal from all others. This function is provided by the multiple user detection (MUD) card 118.

The methods and apparatus described below provide for processing long-code WCDMA at sample rates and can be introduced into a conventional base station as an enhancement to the matched-filter rake receiver. The algorithms and processes can be implemented in hardware, software, or any combination of the two including firmware, field programmable gate arrays (FPGAs), co-processors, and/or array processors.

The following discussion illustrates the calculations involved in the illustrated multiple user detection process. For the following discussion, and as can be recognized by one skilled in the art, the term physical user refers to an actual user. Each physical user is regarded as a composition of virtual users. The concept of virtual users is used to account for both the dedicated physical data channels (DPDCH) and the dedicated physical control channel (DPCCH). There are $1+N_{dk}$ virtual users corresponding to the k^{th} physical user, where N_{dk} is the number of DPDCHs for the k^{th} user.

As one with ordinary skill in the art can appreciate, when long-codes are used, the baseband received signals , $r[t]$, , which is a column vector with one element per antenna, can be modeled as:

$$r[t]=\sum_{k=1}^{K_v}\sum_{m}\tilde{s}_{km}[t-mT_k]b_k[m]+w[t] \qquad (1)$$

where t is the integer time sample index, K_v is the number of virtual users, $T_k=N_kN_c$ is the channel symbol duration, which depends on the user spreading factor, N_k is the spreading factor for the k^{th} virtual user, N_c is the number of samples per chip, $w[t]$ is receiver noise and other-cell interference, $\tilde{s}_{km}[t]$ is the channel-corrupted signature waveform for the k^{th} virtual user over the m^{th} symbol period, and $b_k[m]$ is the channel symbol for the k^{th} virtual user over the m^{th} symbol period.

Since long-codes extend over many symbol periods, the user signature waveform and hence the channel-corrupted signature waveform vary from symbol period to symbol period. For L multi-path components, the channel-corrupted signature waveform for the k^{th} virtual user is modeled as,

$$\tilde{s}_{km}[t]=\sum_{p=1}^{L}a_{kp}s_{km}[t-\tau_{kp}] \qquad (2)$$

where a_{kp} are the complex multi-path amplitudes. The amplitude ratios β_k are incorporated into the amplitudes a_{kp}. One skilled in the art will see that if k and l are virtual users corresponding to the DPCCH and the DPDCHs of the same physical user, then, aside from scaling by β_k and β_l, the amplitudes a_{kp} and a_{lp} are equal. This is due to the fact that the signal waveforms for both the DPCCH and the DPDCH pass through the same channel.

The waveform $s_{km}[t]$ is referred to as the signature waveform for the k^{th} virtual user over the m^{th} symbol period. This waveform is generated by passing the code sequence $c_{km}[n]$ through a pulse-shaping filter $g[t]$,

$$s_{km}[t] = \sum_{r=0}^{N_k-1} g[t - rN_c]c_{km}[r] \qquad (3)$$

where $g[t]$ is the raised-cosine pulse shape. Since $g[t]$ is a raised cosine pulse as opposed to a root-raised-cosine pulse, the received signal $r[t]$ represents the baseband signal after filtering by the matched chip filter. The code sequence $c_{km}[r] \equiv c_k[r + mN_k]$ represents the combined scrambling code, orthogonal variable spreading factor (OVSF) code and j factor associated with even numbered DPDCHs.

The received signal $r[t]$ which has been match-filtered to the chip pulse is next match-filtered by the user long-code sequence filter and combined over multiple fingers. The resulting detection statistic is denoted here as $y_l[m]$, the matched-filter output for the l^{th} virtual user over the m^{th} symbol period. The matched-filter output $y_l[m]$ for the l^{th} virtual user can be written,

$$y_l[m] \equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r[nN_c + \hat{\tau}_{lq} + mT_l] \cdot c_{lm}^{*}[n]\right\} \qquad (4)$$

where $\hat{a}_{lq}^{H}$ is the estimate of a_{lq}^{H}, and $\hat{\tau}_{lq}$ is the estimate of τ_{lq}.

Because of the extreme computational complexity of symbol-rate multiple user detection for long-codes, it is advantageous to resort to regenerative multiple user detection when long-codes are used. Although regenerative multiple user detection operates at the sample rate, for long-codes the overall complexity is lower than with symbol-rate multiple user detection. Symbol-rate multiple user detection requires calculating the correlation matrices every symbol period, which is unnecessary with the signal regeneration methods described herein.

For regenerative multiple user detection, the signal waveforms of interferers are regenerated at the sample rate and effectively subtracted from the received signal. A second pass

through the matched filter then yields improved performance. The computational complexity of regenerative multiple user detection is linear with the number of users.

By way of review, the implementation of the regenerative multiple user detection can be implemented as a baseline implementation. Referring back to the received signal, $r[t]$:

$$\begin{aligned} r[t] &= \sum_{k=1}^{K_v} \sum_{m} \sum_{p=1}^{L} a_{kp} s_{km}[t-\tau_{kp}-mT_k]b_k[m] + w[t] \\ &= \sum_{k=1}^{K_v} r_k[t] + w[t] \end{aligned}$$

$$r_k[t] \equiv \sum_{m} \sum_{p=1}^{L} a_{kp} s_{km}[t-\tau_{kp}-mT_k]b_k[m] \tag{5}$$

For the baseline implementation, all estimated interference is subtracted yielding a cleaned-up signal $\hat{r}_l^{(n+1)}[t]$ as follows:

$$\begin{aligned} \hat{r}_l^{(n+1)}[t] &= r[t] - \sum_{\substack{k=1 \\ k\neq l}}^{K_v} \hat{r}_k^{(n)}[t] \\ \hat{r}_k^{(n)}[t] &\equiv \sum_{m} \sum_{p=1}^{L} \hat{a}_{kp}^{(n)} \cdot s_{km}[t-\hat{\tau}_{kp}^{(n)}-mT_k] \cdot b_k^{(n)}[m] \end{aligned} \tag{6}$$

The implementation represented by Equation (6) corresponds to a total subtraction of the estimated interference. One skilled in the art will appreciate that performance can typically be improved if only a fraction of the total estimated interference is subtracted (i.e., partial interference subtraction), this owing to channel and symbol estimation errors. Equation (6) is easily modified so as to incorporate partial interference cancellation by introducing a multiplicative constant of magnitude less than unity to the sum total of the estimated interference. When multiple cancellation stages are used the optimum value of this constant is different for each stage.

The above equations are implemented in the baseline long-code multiple user detection process 118 as illustrated in Figure 3. The receiver base-band signal $r[t]$ 122 is input to the rake receiver cards 112 (i.e., one rake receiver for each user) as described above. Each of the rake receivers 112 processes the base-band signal $r[t]$ 122 and outputs the first approximation of the transmitted symbol, $y_k^{(0)}[m]$ 304 for each user k (e.g., user 1 through user K), as well as the estimated amplitude $\hat{a}_{kp}^{(0)}$, time lag $\hat{\tau}_{kp}^{(0)}$ and user code 306. For ease of notation, here, the

superscript refers to the n^{th} regeneration iteration. Hence, for example, $\hat{a}_{kp}^{(0)}$ refers to the base-band because no iterations have been performed.

The $y_k^{(0)}[m]$ 304 output from the rake receiver 112 is input into a detector which outputs hard or soft symbol estimates $\hat{b}_k^{(0)}[m]$ used to cancel the effects of multiple access interference (MAI). One skilled in the art will appreciate that many different detectors may be used, including the hard-limiting (sign function) detector, the null-zone detector, the hyperbolic tangent detector and the linear-clipped detector, and that soft detectors (all but the first listed above) typically yield improved performance.

The outputs from the rake receivers 112 and the soft symbol estimates are input into a respreading process 310 which assembles an estimated spread-spectrum waveform corresponding to the selected user but without pulse shaping. The re-spread signals are input into the raised-cosine filter 312 which produces an estimate of the received spread-spectrum waveform for the selected user.

The raised-cosine pulse shaping process accepts the signals from each of the respread processes (e.g., one for each user), and produces the estimated user waveforms $\hat{r}_k^{(n)}[t]$. Next, the waveforms $\hat{r}_k^{(n)}[t]$ are further processed in a series of summation processes 314, 316, 318 to determine each user's cleaned-up signal $\hat{r}_l^{(n+1)}[t]$ according to the above equation (6).

Therefore, for example, to determine the signal corresponding to the 1st user, the base-band signal $r[t]$ 122 from the RF/IF receivers 110 containing information from all simultaneous users is reduced by the estimated signals $\hat{r}_k^{(n)}[t]$ for all users except the 1st user. After the subtraction of the $\hat{r}_k^{(n)}[t]$ signals (e.g., $\hat{r}_2^{(n)}[t]$ through $\hat{r}_{K_v}^{(n)}[t]$ as illustrated), the remainder signal contains predominately the signal for the 1st user. Hence, the summation function 314, applies the above equation (6) to produce the cleaned up signal $\hat{r}_1^{(n+1)}[t]$. This process is performed for each simultaneous user.

The output from the summation processes 314, 316, 318 is supplied to the rake receivers 320 (or re-applied to the original rake receivers 112). The resulting signal produced by the rake receivers 320 is the refined matched-filter detection statistic $y_k^{(1)}[m]$. The superscript (1) indicates that this is the first iteration on the base-band signal. Hence, the base-line long-code multiple user detection is implemented. As illustrated, only one iteration is performed, however, in other embodiments, multiple iterations may be performed depending on limitations (e.g., computational complexity, bandwidth, and other factors).

It can be appreciated by one skilled in the art that the above methods are limited by bandwidth and computational complexity. Specifically, for example, if $K = 128$, i.e., there are 128 simultaneous users for this implementation, the total bisection bandwidth is 998.8 Gbytes/second, determined with the following assumption, for example:

3.84 Mchips / sec / antenna / stream
x 2 antennas
x 8 samples / chip
x 1 bytes / sample
x 128(128 - 1) streams
= 998.8 Gbytes / sec

The computational complexity is calculated in terms of billion operations per second (GOPS), and is calculated separately for each of the processes of re-spreading, raised-cosine filtering, interference cancellation (IC), and the finger receiver operations. The re-spread process involves amplitude-chip-bit multiply-accumulate operations (macs). Assuming, for example, that there are only four possible chips and further that the amplitude chip multiplications are performed via a table look-up requiring zero GOPS, then the re-spread computational complexity is the (amplitude-chip)x(bit macs). Therefore, the re-spread computational cost (in GOPS) is:

3.84 Mchips / sec / antenna / finger / virtual-user / multiple user detection stage
x 2 antennas
x 4 fingers
x 256 virtual users
x 1 multiple user detection stage
x 4 ops / chip (real x complex mac)
= 31.5 GOPS

Based on the same assumptions, the raised-cosine filter requires:

3.84 Mchips / sec / antenna / physical-user / multiple user detection stage
x 8 samples / chip
x 2 antennas
x 128 physical users
x 1 multiple user detection stage
x 6 ops / sample / tap (complex additions then real x complex mac)

x 24 taps (using symmetry)
= 1,132.5 GOPS

The computational cost of the IC process is

3.84 Mchips / sec / antenna / physical-user / multiple user detection stage
x 8 samples / chip
x 2 antennas
x 128 physical users
x 1 multiple user detection stage
x 2 ops / sample / physical users (complex add))
x 128 users
= 2,013.3 GOPS

Finally, the computational complexity for the rake receiver processes is:

3.84 Mchips / sec / antenna / physical-user / multiple user detection stage
x 2 antennas
x 4 fingers
x 256 virtual users
x 1 multiple user detection stage
x 8 ops / chip (complex mac)
= 62.9 GOPS

Summing the separate computational complexities for each of the above processes yields the following results:

Process	GOPS
Re-Spread	31.5
Raised Cosine Filtering	1,132.5
IC	2,013.3
Finger Receivers	62.9
TOTAL	3,240.2

However, both the bandwidth and computation complexity are reduced by employing a residual-signal implementation as now described. The bandwidth can be reduced by forming the residual signal, which is the difference between the received signal and the total (i.e., all

users and all multi-paths) estimated signal. Then, the cleaned-up signal $\hat{r}_l^{(n+1)}[t]$ expressed in terms of the residual signal is:

$$\begin{aligned}\hat{r}_l^{(n+1)}[t] &= r[t] - \sum_{\substack{k=1\\k\neq l}}^{K_v} \hat{r}_k^{(n)}[t]\\ &= \hat{r}_l^{(n)}[t] + r[t] - \sum_{k=1}^{K_v} \hat{r}_k^{(n)}[t]\\ &= \hat{r}_l^{(n)}[t] + r_{res}^{(n)}[t]\end{aligned}$$

$$\begin{aligned}r_{res}^{(n)}[t] &\equiv r[t] - \hat{r}^{(n)}[t]\\ \hat{r}^{(n)}[t] &\equiv \sum_{k=1}^{K_v} \hat{r}_k^{(n)}[t]\end{aligned} \tag{7}$$

This implementation is illustrated in Figure 4. One skilled in the art can recognize that through the point of determining the output from the raised-cosine filters, the residual signal implementation is identical with that above illustrated within Figure 3. It is at this point, the residual signal implementation varies as now described.

A summation process 402 calculates $r_{res}^{(n)}[t]$ according to equation (7) above by accepting the base-band signal $r[t]$ and subtracting the signal $\hat{r}^{(n)}[t]$ (i.e., the output from all of the raised-cosine filters 310).

Differing from the baseline implementation, here, a first summation process 402 is performed by subtracting from the baseband signal $r[t]$ 122 the output from each raised-cosine pulse shaping process 310. This produces the residual signal $r_{res}^{(n)}[t]$ corresponding to the baseband signal and the total (e.g., all users in all multi-paths) estimated signal.

The residual signal $r_{res}^{(n)}[t]$ is supplied to a further summation process for each user (e.g., 404) where the output from that user's raised-cosine pulse shaping process 312 is added to the $r_{res}^{(n)}[t]$ signal as described in above equation (7), thus determining the cleaned-up signal $\hat{r}_l^{(n+1)}[t]$ for each user.

Next, as with the baseline implementation, the cleaned-up signal $\hat{r}_l^{(n+1)}[t]$ for each user is supplied to a rake receiver 320 (or reapplied to 112) for processing into the resultant $y_l^{(n+1)}[m]$ detection statistics ready for processing by the symbol processor 120.

One skilled in the art can recognize that both the bandwidth and computational complexity is improved (i.e., lowered) for this implementation compared with the base-line implementation described above. Specifically, continuing with the assumptions used in determining the bandwidth and computational complexity as above and applying those assumptions to the residual-signal implementation, the bandwidth can be estimated as follows:

3.84 Mchips / sec / antenna / stream
x 2 antennas
x 8 samples / chip
x 1 bytes / sample
x 129 streams
= 7.9 Gbytes / sec

The computational complexity for each of the processes is as follows: the re-spreading and raised-cosine are the same as with the baseline implementation.

For the IC processes, the computational complexity is:

3.84 Mchips / sec / antenna / physical-user / multiple user detection stage
x 8 samples / chip
x 2 antennas
x 128 physical users
x 1 multiple user detection stage
x 2 ops / sample / waveform addition (complex add))
x 3 waveform additions
= 47.2 GOPS

Finally, the finger receiver processes are the same as with the base-line implementation above. Therefore, summing the separate computational complexities for each of the above processes yields the following results:

Process	GOPS
Re-Spread	31.5
Raised Cosine Filtering	1,132.5
IC	47.2
Finger Receivers	62.9
TOTAL	1,274.1

Therefore, both the bandwidth and computational complexity is improved, however, it can be recognized by one skilled in the art that even with such improvement, the computational complexity may be a limiting factor.

Further improvement is possible and is now described within in the following three embodiments, although other embodiments can be recognized by one skilled in the art. One improvement is to utilize a implicit waveform subtraction rather than the explicit waveform subtraction described for use with both the baseline implementation and the residual long-code implementation above. A considerable reduction in computational complexity results if the individual user waveforms are not explicitly calculated, but rather implicitly calculated.

The illustrated embodiment utilize implicit waveform subtraction by expanding on equation (7) above, and using approximations as shown below in equation (8).

$$\begin{aligned}
y_l^{(n+1)}[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\frac{1}{2N_l}\sum_{n=0}^{N_l-1}\hat{r}_l^{(n+1)}[nN_c+\hat{\tau}_{lq}^{(n)}+mT_l]c_{lm}^{*}[n]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\frac{1}{2N_l}\sum_{n=0}^{N_l-1}\hat{r}_l^{(n)}[nN_c+\hat{\tau}_{lq}^{(n)}+mT_l]c_{lm}^{*}[n]\right\} \\
&+\mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\cdot\frac{1}{2N_l}\sum_{n=0}^{N_l-1}r_{res}^{(n)}[nN_c+\hat{\tau}_{lq}^{(n)}+mT_l]c_{lm}^{*}[n]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\cdot\frac{1}{2N_l}\sum_{n=0}^{N_l-1}\left[\sum_{m'}\sum_{q'=1}^{L}\hat{a}_{lq'}^{(n)}s_{lm'}[nN_c+\hat{\tau}_{lq}^{(n)}-\hat{\tau}_{lq'}^{(n)}+(m-m')T_l]\hat{b}_l^{(n)}[m']\right]c_{lm}^{*}[n]\right\}+y_{res,l}^{(n)}[m] \\
&\cong \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\frac{1}{2N_l}\sum_{n=0}^{N_l-1}\left[\sum_{q'=1}^{L}\hat{a}_{lq'}^{(n)}s_{lm}[nN_c+\hat{\tau}_{lq}^{(n)}-\hat{\tau}_{lq'}^{(n)}]\right]c_{lm}^{*}[n]\right\}\cdot\hat{b}_l^{(n)}[m]+y_{res,l}^{(n)}[m] \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L}\sum_{q'=1}^{L}\hat{a}_{lq}^{(n)H}\hat{a}_{lq'}^{(n)}\frac{1}{2N_l}\sum_{n=0}^{N_l-1}s_{lm}[nN_c+\hat{\tau}_{lq}^{(n)}-\hat{\tau}_{lq'}^{(n)}]c_{lm}^{*}[n]\right\}\hat{b}_l^{(n)}[m]+y_{res,l}^{(n)}[m] \\
&\cong \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\hat{a}_{lq}^{(n)}\right\}\cdot\hat{b}_l^{(n)}[m]+y_{res,l}^{(n)}[m] \\
&= A_l^{(n)2}\cdot\hat{b}_l^{(n)}[m]+y_{res,l}^{(n)}[m]
\end{aligned}$$

$$\begin{aligned}
A_l^{(n)2} &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\hat{a}_{lq}^{(n)}\right\} \\
y_{res,l}^{(n)}[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{(n)H}\cdot\frac{1}{2N_l}\sum_{n=0}^{N_l-1}r_{res}^{(n)}[nN_c+\hat{\tau}_{lq}^{(n)}+mT_l]\cdot c_{lm}^{*}[n]\right\}
\end{aligned} \quad (8)$$

The two approximations used, as indicated within equation (8), include neglecting inter-symbol interference terms for the user of interest, and further, neglecting cross-multi-path interference terms for the user of interest. Because the user of interest term has a strong deterministic term, the omission of these low-level random contributions is justified. These contributions could be included in a more detailed embodiment without incurring excessive increases in computational complexity. However, implementation computational complexity would increase somewhat. Such an embodiment may be appropriate for high data-rate, low spreading factor users where inter-symbol and cross multi-path term are larger.

A noteworthy aspect of equation (8) above is that the rake receiver operation on the estimated user of interest signal $\hat{r}_l^{(n)}[t]$ can be calculated analytically. Thus, the signal need not be explicitly formed, but rather, the corresponding contribution is added after the rake receiver operation on the residual signal alone. Now referring to Figure 5, this implicit waveform subtraction implementation is illustrated.

One skilled in the art can glean from the illustration that separate re-spreading and raised-cosine processing is no longer performed on each individual user signal, but rather, is performed only once on the baseband composite re-spread signal $\rho^{(n)}[t]$. Thus, the re-spread process 312 accumulates the composite signal $\rho^{(n)}[t]$ based on the amplitudes $\hat{a}_{kp}^{(n)}$, time lags $\hat{\tau}_{kp}^{(n)}$ and user codes. The output from the re-spreading process produces another composite signal $\hat{r}^{(n)}[t]$ 502 as described below and in equation (9).

At this point, it is of note that a substantial reduction in computational complexity accrues due to not having to explicitly calculate the individual user estimated waveforms. As illustrated in Figure 5, the individual user waveforms are not required, hence, the composite signal $\rho^{(n)}[t]$ 502 representing the sum of all estimated user waveforms can be formed by calculating this composite waveform first without performing the raised-cosine filtering process on each individual waveform. Only one filtering operation need be performed, which represents a substantial reduction in computational complexity.

The form of $\rho^{(n)}[t]$ is as follows:

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v} \sum_{p=1}^{L} \sum_{r} \delta[t - \hat{\tau}_{kp}^{(n)} - rN_c] \cdot \hat{a}_{kp}^{(n)} \cdot c_k[r] \cdot \hat{b}_k^{(n)}[\lfloor r / N_k \rfloor]$$

$$\hat{r}^{(n)}[t] = \sum_{r} g[r] \rho^{(n)}[t-r] \tag{9}$$

Now that an understanding of the composite waveform $\rho^{(n)}[t]$ is accomplished, referring back to Figure 5, this waveform is transformed into $\hat{r}^{(n)}[t]$ via the raised-cosine pulse shaping filter 312. From here, a summation process 506 subtracts $\hat{r}^{(n)}[t]$ from the base-line waveform $r[t]$ producing the residual waveform $r_{res}^{(n)}[t]$ as shown above (e.g., in equation (7)).

Unlike the residual signal implementation described above, here, the $r_{res}^{(n)}[t]$ is applied directly to the rake receivers 506 (or reapplied to the rake receivers 112) for each user together with the user code for that user. The output from each rake receiver is applied to a summation process, where the $A_l^{(n)2} \cdot \hat{b}_l^{(n)}[m]$ values are added to the rake receiver output as described above in equation (8) producing the $y_l^{(n+1)}[m]$ detection statistics suitable for symbol processing 120.

The computational complexity of this embodiment is reduced as now described. The re-spread processing and rake receiver computational costs are the same as with the previous implementations. However, the raise-cosine filtering and interference cancellation computational cost is now:

For the raised-cosine filtering,

3.84 Mchips / sec / antenna / multiple user detection stage
x 8 samples / chip
x 2 antennas
x 1 multiple user detection stage
x 6 ops / sample (complex addition then real x complex mac)
x 24 taps (using symmetry)
= 8.8 GOPS
The computational cost of the IC process is

3.84 Mchips / sec / antenna / multiple user detection stage
x 8 samples / chip
x 2 antennas
x 1 multiple user detection stage
x 2 ops / sample / waveform addition (complex add)
x 1 waveform addition
= 0.123 GOPS

Summing the separate computational complexities for each of the above processes yields the following results:

Process	GOPS
Re-Spread	31.5
Raised Cosine Filtering	8.8
IC	0.1
Finger Receivers	62.9
TOTAL	103.3

Another embodiment using matched-filter outputs rather than antenna streams is now presented. This embodiment follows from equation (8) above where the rake receiver outputs are:

$$y_{res,l}^{(n)}[m] \equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r_{res}^{(n)}[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \tag{10}$$

and further user equation (7) above, equation (10) can be re-written as:

$$\begin{aligned} y_{res,l}^{(n)}[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r_{res}^{(n)}[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \\ &= \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \\ &\quad - \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \hat{r}^{(n)}[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \\ &= y_l^{(n)}[m] - \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \hat{r}^{(n)}[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \end{aligned} \tag{11}$$

and then, combining equation (11) with equation (8) yields:

$$y_l^{(n+1)}[m] = A_l^{(n)2} \cdot \hat{b}_l^{(n)}[m] + y_l^{(n)}[m] - \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n)H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \hat{r}^{(n)}[nN_c + \hat{\tau}_{lq}^{(n)} + mT_l] \cdot c_{lm}^{*}[n]\right\} \tag{12}$$

This embodiment improves the above approaches in that the antenna streams do not need to be input into the multiple user detection process, however, it is not possible to re-estimate the channel amplitudes.

Referring to Figure 6, an illustration of the matched-filter output embodiment is illustrated. As illustrated, the processing of the baseband $r[t]$ waveform is accomplished as

described in Figure 5 above, and further, $\rho^{(n)}[t]$ is determined in accordance with equation (9) and is applied to the raised-cosine pulse shaping process 602.

Differing from the above embodiment, however, there is no summation process before applying $\hat{r}^{(n)}[t]$ of the second rake receiver process 604. Rather, $\hat{r}^{(n)}[t]$ is applied directly to the rake receiver process 604. The output from the rake receivers 604 is subtracted 606 from the output $y_l^{(n)}[m]$ from the first rake receivers 112. This difference is then added to the $A_l^{(n)2} \cdot \hat{b}_l^{(n)}[m]$ value to produce $y_l^{(n+1)}[m]$. This process is described within the above equations (11) and (12).

The computational complexity is reduced because there is no longer an explicit interference canceling (IC) operation, and thus, the interference canceling computational cost is zero. The rake receiver computational cost is half the previous embodiment's value because now the re-estimate of the amplitudes cannot be performed, and there is no need to cancel interference on the dedicated physical control channel (DPCCH). Therefore, the computational cost is:

Process	GOPS
Re-Spread	31.5
Raised Cosine Filtering	8.8
IC	0.0
Finger Receivers	31.5
TOTAL	71.8

Another embodiment using matched-filter outputs obtained before the maximal ratio combination (MRC) is now described. The pre-MRC rake matched-filter outputs can be described as:

$$y_{lq}^{(0)}[m] = \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l] \cdot c_{lm}^{*}[n] \qquad (13)$$

The same detection statistics based on the cleaned up signal $\hat{r}_l^{(n+1)}[t]$ is

$$y_{lq}^{(n+1)}[m] = \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \hat{r}_l^{(n+1)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l] \cdot c_{lm}^{*}[n] \qquad (14)$$

Now from Equation (7),

$$\hat{r}_l^{(n+1)}[t] = \hat{r}_l^{(n)}[t] + r[t] - \hat{r}^{(n)}[t] \qquad (15)$$

Hence the first-stage pre-MRC matched-filter outputs can be re-written:

$$y_{lq}^{(n+1)}[m] = \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}_l^{(n+1)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n]$$

$$= \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}_l^{(n)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n]$$

$$+ \frac{1}{2N_l}\sum_{n=0}^{N_l-1} r[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n]$$

$$- \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}^{(n)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n]$$

$$= \hat{a}_{lq}^{(n)}\cdot \hat{b}_l^{(n)}[m] + y_{lq}^{(n)}[m] - y_{est,lq}^{(n)}[m]$$

$$y_{est,lq}^{(n)}[m] \equiv \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}^{(n)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n] \quad (16)$$

where the following approximation has been used,

$$y_{lq,1}^{(n+1)}[m] \equiv \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}_l^{(n)}[nN_c + \hat{\tau}_{lq}^{(0)} + mT_l]\cdot c_{lm}^*[n] \cong \hat{a}_{lq}^{(n)}\cdot \hat{b}_l^{(n)}[m] \quad (17)$$

Given the pre-MRC matched-filter outputs the re-estimated channel amplitudes are

$$\hat{a}_{lq}^{(n+1)} \equiv \sum_s w[s]\cdot \frac{1}{N_p}\sum_{m=0}^{N_p-1} y_{lq}^{(n+1)}[m+Ms]\cdot \hat{b}_l^{(n)}[m+Ms] \quad (18)$$

wherein

$w[s]$ is a filter,

N_p is a number of symbols, and

M is a number of symbols per slot,

and the post-MRC matched-filter outputs are then

$$y_l^{(n+1)}[m] \equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{(n+1)^H} \cdot y_{lq}^{(n+1)}[m]\right\} \tag{19}$$

This embodiment is illustrated in Figure 7. Here, the $y_{lq}^{(0)}[m]$ detection statistics are produced as with the above embodiments, however, before being applied to the MRC, the estimated amplitude $\hat{a}_{lq}^{(0)}$ is determined first. Next, the MRC produces the $y_l^{(0)}[m]$ detection statistics which are from the amplitudes $\hat{a}_{lq}^{(0)}$ and the pre-combination matched-filter detection statistics $y_{lq}^{(0)}[m]$ as in Equation (19) above.

The $\hat{r}[t]$ waveform is applied (or reapplied) to a rake receiver 704. The output from the rake receiver 704 is subtracted 706 from the $y_{lq}^{(0)}[m]$ detection statistics. Next, the difference from the subtraction 706 is summed 708 with the $\hat{a}_{lq}^{(0)} \cdot \hat{b}_l^{(0)}[m]$ value, thus producing $y_{lq}^{(1)}[m]$ in accordance with equation (19) above.

After n iterations are performed, the $y_{lq}^{(n)}$ detection statistics for each of the users corresponding to each antenna has been determined. The detection statistics for each user, $y_l^{(n)}$ is next determined via estimating the complex amplitudes 710 across the Q channels for that user, and performing a maximum ratio combination 712 using those amplitudes.

It is helpful to understand that although the computational complexity increased, here, it is possible to re-estimate channel amplitudes, and hence, cancel interference on the dedicated physical control channels (DPCCH). The computational complexity of this embodiment is:

Process	**GOPS**
Re-Spread	31.5
Raised Cosine Filtering	8.8
IC	0.0
Finger Receivers	62.9
TOTAL	103.2

which is still within a practical range.

Therefore, as shown in all the embodiments above, and other non-illustrated embodiments, methods for performing multiple user detection are illustrated.

Turning now to software implementations for the above, one of several implementations is designed to allow full or partial decoding of users at various transmission time intervals (TTIs) within the multiple user detection (MUD) iterative loop. The approach, illustrated in

Figure 8, allows users belonging to different classes (e.g., voice and data) to be processed with different latencies. For example, voice users could be processed with a 10+ ms latency 802, whereas data users could be process with an 80+ ms latency 804. Alternately, voice users could be processed with a 20+ms latency 806 or a 40+ ms latency 808, so as to include voice decoding in the MUD loop. Other alternatives are possible depending on the implementation and limitations of the processing requirements.

If a particular data user is to be processed with an 80+ ms latency 804 so as to include the full turbo decode within the MUD loop then the input channel bit-error rate (BER) pertaining to these users might be extraordinarily high. Here, the MUD processing might be configured so as to not include any cancellation of the data users within the 10+ ms latency 802. These data users would then be cancelled in the 20+ ms latency 806 period. For this cancellation it could be opted to perform MUD only on data users. The advantage of canceling the voice users in the first latency range (e.g., first box) would still benefit the second latency range processing.

Alternately, the second box 806 could perform cancellation on both voice and data users. The reduced voice channel bit-error rate would not benefit the voice users, whose data has already been shipped out to meet the latency requirement, but the reduced voice channel BER would improve the cancellation of voice interference from the data users. In the case that voice and data users are cancelled in the second box 806, another, possible configuration would be to arrange the boxes in parallel. Other reduced-latency configurations with mixed serial and parallel arrangements of the processing boxes are also possible.

Depending on the arrangement chosen, the performance for each class of user will vary. The approach above tends to balance the propagation range for data and voice users, and the particular arrangement can be chosen to tailor range for the various voice and data services.

Each box is the same code but configured differently. The parameters that differ are:

N_FRAMES_RAKE_OUTPUT;
Decoding to be performed (e.g. repetition decoding, turbo decoding, and the like);
Classes of users to be cancelled;
Threshold parameters.

The pseudo code for the software implementation of one long-code multiple user detection processing box is as follows:

```
Initialize
        Zero data
        Generate OVSF codes
                Generate raised cosine pulse
        Allocate memory


        Open rake output files
        Open mod output files
        Align mod data

Main Frame Loop {
        Determine number of physical users
        Read_in_rake_output_records (N frames)
        Reformat_rake_output_data (N frames at a time)
        for stage = 1 : N_stages
                Perform appropriate decoding(SRD, turbo, and the like, depending on TTI)
                Perform_long_code_mud
        end
}
Free memory
```

The following four functions are described below:

Read_in_rake_output_records;
Reformat_rake_output_data
Perform appropriate decoding(SRD, turbo, and the like, ``depending on TTI);
Perform_long_code_mud.

The Read_in_rake_output_records function performs:
Reading in data for each user; and
Assigning data structure pointers.

The rake data transferred to MUD is associated with structures of type Rake_output_data_type. The elements of this structure are given in Table 1. There is a parameter N_FRAMES_RAKE_OUTPUT with values { 1, 2, 4, 8 } that specifies the number of frames to be read-in at a time. The following table tabulates the Structure Rake_output_buf_type elements:

Element Type	Name
unsigned long	Frame_number
unsigned long	physical_user_code_number

int	physical_user_tfci
int	physical_user_sf
int	physical_user_beta_c
int	physical_user_beta_d
int	N_dpdchs
int	compressed_mode_flag
int	compressed_mode_frame
int	N_first
int	TGL
int	slot_format
int	N_rake_fingers
int	N_antennas
unsigned long	mpath_offset[N_ANTENNAS]
unsigned long	tau_offset
unsigned long	y_offset
COMPLEX*	mpath[N_ANTENNAS]
unsigned long*	tau_hat
float *	y_data

It is helpful to describe several structure elements for a complete understanding. The element slot_format is an integer from 0 to 11 representing the row in the following table (DPCCH fields), 3GPP TS 25.211. By way of non-limiting example, when slot_format = 3, it maps to the fourth row in the table corresponding to slot format 1 with 8 pilot bits and 2 TPC bits. The offset values (e.g. tau_offset) give the location in memory relative to the top of the structure where the corresponding data is stored. These offset values are used for setting the corresponding pointers (e.g. tau_hat). For example, if Rbuf is a pointer to the structure then:

```
Rbuf->tau_hat = (unsigned long*)( (unsigned long)Rbuf + Rbuf->tau_offset );
```

is used to set the tau_hat pointer.

The rake output structure associated data (mpath, tau_hat and y_data) is ordered as follows:

mpath[n][q + s * L]	= amplitude data
tau_hat[q]	= delay data
y_data[0 + m * M]	= DPCCH data for symbol period m
y_data[1+j+(d-1)*J + m * M]	= dth DPDCH data for symbol period m

where

n	= antenna index (0 : Na-1)
q	= finger index (0 : L-1)
s	= slot index (0 : Nslots-1)
m	= symbol index (0 : 149)
j	= bit index (0 : J-1)
d	= DPDCH index (1 : Ndpdchs)
Na	= N_ANTENNAS
L	= N_RAKE_FINGERS_MAX
Nslots	= N_SLOTS_PER_FRAME = 15
J	= 256 / SF
M	= 1 + J * Ndpdchs.

The memory required for the rake output buffers is dominated by the y-data memory requirement. The maximum memory requirement for a user is Nsym * (1 + 64 * 6) floats per frame, where Nsym = 150 is the number of symbols per frame. This corresponds to 1 DPCCH at SF 256 and 6 DPDCHs at SF 4. If 128 users are all allocated to this memory then possible memory problems arise. To minimize allocation problems, the following table gives the maximum number of user that the MUD implementation will be designed to handle at a given SF and Ndpdchs.

SF	Ndpdchs	Number users	Bits per symbol	Mean bits per symbol
256	1	256	2	4.0
128	1	192	3	4.5
64	1	128	5	5.0
32	1	96	9	6.8
16	1	64	17	8.5
8	1	32	33	8.3
4	1	16	65	8.1
4	2	12	129	12.1
4	3	8	193	12.1
4	4	4	257	8.0
4	5	3	321	7.5
4	6	2	385	6.0

In the proceeding table, the Bits per symbol = 1 + (256 / SF) * N_DPDCHs, Mean bits per symbol = (Number users) * (Bits per symbol) / 128, and Ndpdchs = Number DPDCHs.

From the above table it is noted that the parameter specifying the mean number of bits per symbol be set to MEAN_BITS_PER_SYMBOL = 16. The code checks to see if the physi-

cal users specifications are consistent with this memory allocation. Given this specification, the following are estimates for the memory required for the rake output buffers.

Data	Type	Size	Count	Count	Bytes
Rake_output_buf	Structure	88	1	1	88
mpath	COMPLEX	8	Lmax * Nslots * Na	240	1,920
tau	int	4	Lmax	8	32
y	float	4	Nsym * Nbits	2400	9,600
y_{Iq}	COMPLEX	8	Nsym * Nbits * Lmax * Na		307,200

Total bytes per user per frame 318,840

Total bytes for 128 users and 9 frames 367 Mbytes

Where Count is the per physical user per frame, assuming numeric values based on:

Lmax = N_RAKE_FINGERS_MAX = 8
Nslots = N_SLOTS_PER_FRAME = 15
Na = N_ANTENNAS = 2
Nsym = N_SYMBOLS_PER_FRAME = 150
Nbits = MEAN_BITS_PER_SYMBOL = 16

The location of each structure is stored in an array of pointers

Rake_output_buf[User + Frame_idx * N_USERS_MAX]

where Frame_idx varies from 0 to N_FRAMES_RAKE_OUTPUT inclusive. Frame 0 is initially set with zero data. After all frames are processed, the structure and data corresponding to the last frame is copied back to frame 0 and N_FRAMES_RAKE_OUTPUT new structures and data are read from the input source.

The Reformat_rake_output_data function performs:

Combining of multi-path data across frame boundaries;
Determines number of rake fingers for each MUD processing frame
Filling virtual-user data structures
Separates DPCHs into virtual users
Determines chip and sub-chip delays for all fingers
Determines the minimum SF and maximum number of DPDCHs for each user

Reformats user b-data to correspond to the minimum SF
Reformats rake data to be linear and contiguous in memory.

Interference cancellation is performed over MUD processing frames. Due to multi-path and asynchronous users, the MUD processing frame will not correspond exactly with the user frames. MUD processing frames, however, are defined so as to correspond as closely as possible to user frames. It is preferable for MUD processing that the number of multi-path returns be constant across MUD processing frames. The function of multi-path combining is to format the multi-path data so that it appears constant to the long-code MUD processing function. Each time after N = N_FRAMES_RAKE_OUTPUT frames of data is read from the input source the combining function is called.

Figure 9 shows a hypothetical set of multi-path lags corresponding to several frames of user data 902. Also shown are the corresponding MUD processing frames 904. Notice that MUD processing frame k overlaps with user frames k-1 and k. For example, processing frame 1 906 overlaps with user frame 0 908, and further, overlaps with user frame 1 910. The MUD processing frame is positioned so that this is true for all multi-paths of all users. A one-symbol period corresponds to a round trip for a 10 km radius cell. Hence even large cells are typically only a few symbols asynchronous.

The multi-path combining function determines all distinct delay lags from user frames k-1 and k. Each of these lags is assigned as a distinct multi-path associated with MUD processing frame k, even if some of the distinct lags are obviously the same finger displaced in delay due to channel dynamics. The amplitude data for a finger that extends into a frame where the finger wasn't present is set to zero. The illustrated thin lag-lines (e.g., 912) represent finger amplitude data that is set to zero. After the tentative number of fingers is assessed in this way, the total finger energy that falls within the MUD processing frame is assessed for each tentative finger and the top N_RAKE_FINGERS_MAX fingers are assigned. In the assignment of fingers the finger indices for fingers that were active in the previous MUD processing frame are kept the same so as not to drop data.

The user SF and number of DPDCHs can change every frame. It is helpful for efficient MUD processing that the user SF and number of DPDCHs be constant across MUD processing frames. This function, Reformat_rake_output_data formats the user b-data so that it appears constant to the long-code MUD processing function. Each time after N = N_FRAMES_RAKE_OUTPUT frames of data is read from the input source this function is called. The function scans the N frames of rake output data and determined for each user the minimum SF and maximum number of DPDCHs. Virtual users are assigned according to the maximum

number of DPCHs. If for a given frame the user has fewer DPCH the corresponding b-data and a-data are set to zero.

Note that this also applies to the case where the number of DPDCHs is zero due to inactive users, and also to the case where the number of DPCHs is zero due to compressed mode. It is anticipated that the condition of multiple DPDCHs will not often arise due to the extreme use of spectrum. If for a given frame the SF is greater than the minimum the b-data is expanded to correspond to the lower SF. That is, for example, if the minimum SF is 4, but over some frames the SF is 8, then each SF-8 b-data bit is replicated twice so as to look like SF-4 data. Before the maximum ration combination (MRC) operation the y-data corresponding to expanded b-data is averaged to yield the proper SF-8 y-data.

Figure 10 shows how rake output data is mapped to (virtual) user data structures. Each small box (e.g., 1002) in the figure represents a slot's-worth of data. For DPCCH y-data or b-data, for example, each box would represent 150 values. Data is mapped so as to be linear in memory and contiguous frame to frame for each antenna and each finger. The reason for this mapping is that data can easily be accessed by adjusting a pointer. A similar mapping is used for other data except the amplitude data, where it would be imprudent to attempt to keep the number of fingers constant over a time period of up to 8 frames. For the virtual-user code data there are generally 38,400 data items per frame; and for the b-data and y-data there are generally 150 x 256 / SF data items per frame.

Note that for pre-MRC y-data, the mapping is linear and contiguous in memory for each antenna and each finger. Each DPCH is mapped to a separate virtual user data structure. The initial conditions data (frame 0 1004) is initially filled with zero data (except for the codes). After frame N data is written, this data is copied back to frame 0 1004, and the next frame of data that is written is written to frame 1 1006. For all data types the 0-index points to the first data item written to frame 0 1004. For example, the initial-condition b-data (frame 0) for an SF 256 virtual user is indexed b[0], b[1], ..., b[149], and the b-data corresponding to frame 1 is b[150], b[151], ..., b[299].

Four indices are of interest: chip index, bit index, symbol index, and slot index. The chip index r is always positive. All indices are related to the chip index. That is, for chip index r we have

Chip index	= r
Bit index	= r / Nk
Symbol index	= r / 256

Slot index = r / 2560

where Nk is the spreading factor for virtual user k.

The elements for the (virtual) user data structures are given in the following table along with the memory requirements.

Element Type	Name	Bytes	Bytes
int	Dpch_type	4	4
int	Sf	4	4
int	log2Sf	4	4
float	Beta	4	4
int	Mrc_bit_idx	4	4
int	N_bits_per_dpch	4	4
int	N_rake_fingers[Nf]	4*8	32
int	Chip_idx_rs[Lmax]	4*8	32
int	Chip_idx_ds[Lmax]	4*8	32
int	Delay_lag[Lmax]	4*8	32
int	finger_idx_max_lag	4	4
int	Chip_delay[Lmax]	4*8	32
int	Sub_chip_delay[Lmax]	4*8	32
COMPLEX	axcode[Nf][Na][Lmax][Nslots * 2][4]	8*8*2*8*15*2*4	122880
COMPLEX	a_hat_ds[Nf][Na][Lmax][Nslots * 2]	8*8*2*8*15*2	30720
COMPLEX*	mf_ylq[Na][Lmax]	4*2*8	64
COMPLEX*	mud_ylq[Na][Lmax]	4*2*8	64
float*	mf_y_data	4	4
float*	mud_y_data	4	4
char*	mf_b_data	4	4
char*	mud_b_data	4	4
char*	mod_b_data	4	4
char	Code[Nchips * (1+Nf)]	1*38400*9	345600
COMPLEX	mud_ylq_save[Na][Lmax]	8*2*8	128
int	Mrc_bit_idx_save	4	4
float	Repetition_rate	4	4
COMPLEX1,2	mf_ylq[Na][Lmax][Nbits1 * (1+Nf)]	8*2*8*1200*9	1382400
COMPLEX1,2	mud_ylq[Na][Lmax][Nbits1 *(1+Nf)]	8*2*8*1200*9	1382400
float1,2	mf_y_data[Nbits1 * (1+Nf)]	4*1200*9	43200
float1,2	mud_y_data[Nbits1 * (1+Nf)]	4*1200*9	43200
char(1,2)	mf_b_data[Nbits1 * (1+Nf)]	1*1200*9	10800

char(1,2)	mud_b_data[Nbits1 * (1+Nf)]	1*1200*9	10800
char(1,2)	mod_b_data[Nbits1 * (1+Nf)]	1*1200*9	10800
Total			3,383,304
x 256 v-users			866 Mbytes
OLD:			
COMPLEX	Code[Nchips * 2]	8*38400*2	614400

where the following notations are defined:

1 - Associated data, not explicitly part of structure
2 - Based on 8 bits per symbol on average

Lmax	= N_RAKE_FINGERS_MAX	= 8
Na	= N_ANTENNAS	= 2
Nslots	= N_SLOTS_PER_FRAME	= 15
(Nbitsmax1	= N_BITS_PER_FRAME_MAX_1	= 9600)
Nchips	= N_CHIPS_PER_FRAME	= 38400
Nf	= N_FRAMES_RAKE_OUTPUT	= 8
Nbits1	= MEAN_BITS_PER_FRAME_1	= 150*4.25 ~= 640.

Each user class has a specified decoding to be performed. The decoding can be:

None
Soft Repetition Decoding (SRD)
Turbo decoding
Convolutional decoding.

All decoding is Soft-Input Soft-Output (SISO) decoding. For example, an SF 64 voice user produces 600 soft bits per frame. Thus 1,200 soft bits per 20 ms transmission time intervals (TTIs) are produced. These 1,200 soft bits are input to a SISO de-multiplex and convolution decoding function that outputs 1,200 soft bits. The SISO de-multiplex and convolution decoding function reduces the channel bit error rate (BER) and hence improve MUD performance. Since data is linear in memory no reformatting of data is necessary and the operation can be performed in-place. If further decoders are included, reduced complexity partial-decode variants can be employed to reduce complexity. For turbo decoding, for example, the number of iterations may be limited to a small number.

The Long-code MUD performs the following operations:

Respread
Raised-Cosine Filtering
Despread
Maximal-Ratio Combining (MRC).

The re-spread function calculates r[t] given by

$$\rho[t] \equiv \sum_{k=0}^{K_v-1} \sum_{p=0}^{L-1} \sum_{r} \delta[t-\hat{\tau}_{kp}-rN_c] \cdot \hat{a}_{kp}[r/2560] \cdot c_k[r] \cdot \hat{b}_k[r/N_k] \tag{20}$$

The function r[t] is calculated over the interval t = 0 : Nf*M*Nc - 1, where M = 38400 is the number of chips per frame and Nf is the number of frames processed at a time. The actual function calculated is

$$\rho_m[t] \equiv \rho[t + mN_c N_{chips}]$$

$$t = 0 : N_c N_{chips} - 1 \tag{21}$$

which represents a section of the waveform of length Nchips chips, and the calculation is performed for m = 0 : Nf*M*Nc / Nchips - 1 . The function is defined (and allocated) for negative indices - (Lg -1) : -1, representing the initial conditions which are set to zero at start-up. The parameter Lg is the length of the raised-cosine filter discussed below.

Note that every finger of every user adds one and only one non-zero contribution per chip within this interval corresponding to chip indices r. Given the delay lag tlq for the qth finger of the lth user we can determine which chip indices r contribute to a given interval. To this end define

$$\begin{aligned} t &= nN_c + q, & 0 \le q < N_c \\ \hat{\tau}_{kp} &\equiv n_{kp} N_c + q_{kp}, & 0 \le q_{kp} < N_c \end{aligned} \tag{22}$$

The first definition defines t as belonging to the nth chip interval; the second is a decomposition of the delay lag into chip delay and sub-chip delay. Given the above we can solve for r and q using

$$r = n - n_{kp}$$
$$q = q_{kp} \tag{23}$$

Notice that chip indices r as given above can be negative. In the implementation the pointers $\hat{a}_{kp}$, c_k and $\hat{b}_k$ point to the first element of frame 1 1006 (Figure 10).

The repeated amplitude-code multiplies are avoided by using:

$$(\hat{a}\cdot c)_{kp}[s][\mathbf{c}_k[r]] \equiv \hat{a}_{kp}[s]\cdot c_k[r]$$

$$(\hat{a}\cdot c)_{kp}[s][c] \equiv \begin{cases} \hat{a}_{kp}[s]\cdot(+1+j), & c=0 \\ \hat{a}_{kp}[s]\cdot(-1+j), & c=1 \\ \hat{a}_{kp}[s]\cdot(-1-j), & c=2 \\ \hat{a}_{kp}[s]\cdot(+1-j), & c=3 \end{cases}$$

$$\mathbf{c}_k[r] \equiv \begin{cases} 0, & c_k[r] = +1+j \\ 1, & c_k[r] = -1+j \\ 2, & c_k[r] = -1-j \\ 3, & c_k[r] = +1-j \end{cases} \tag{24}$$

The raised-cosine filtering operation applied to the re-spread signal r[t] produces an estimate of the received signal given by:

$$\hat{r}[t] = \sum_{t'=0}^{L_g-1} g[t']\cdot \rho[t-t'] \tag{25}$$

where g[t] is the raised-cosine pulse and

t = 0 : Nc*Nchips - 1

t' = 0 : Lg - 1

Lg = Nsamples-rc (length of raised-cosine filter)

For example, if an impulse at t = 0 is passed through the above filter the output is g[t]. The position of the maximum of the filter then specifies the delay through filter. The delay is

relevant since it specifies the synchronization information necessary for subsequent despreading. The raised cosine filter is calculated over the time period n = (n1 : n2) / Nc, where Nc is the number of samples per chip, and time is in chips. Note that n1 is negative, and the position of the maximum of the filter is at n = 0. The length of the filter is then Lg = n2 - n1, and the maximum occurs at sample n1. The delay is thus n1 samples, and the chip delay is n1 / Nc chips. For simplicity of implementation n1 is required to be a multiple of Nc.

The de-spread operation calculates the pre-MRC detection statistics corresponding to the estimate of the received signal:

$$y^{(1)}_{est,lq}[m] \equiv \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \hat{r}[nN_c + \hat{\tau}_{lq} + mT_l] \cdot c^{*}_{lm}[n] \tag{26}$$

Prior to the MRC operation, the MUD pre-MRC detection statistics are calculated according to:

$$y^{(1)}_{lq}[m] = \hat{a}_{lq} \cdot \hat{b}_l[m] + y^{(0)}_{lq}[m] - y^{(1)}_{est,lq}[m] \tag{27}$$

These are then combined with antenna amplitudes to form the post-MRC detection statistics:

$$y^{(1)}_l[m] \equiv \mathrm{Re}\left\{ \sum_{q=1}^{L} \hat{a}^{(1)H}_{lq} \cdot y^{(1)}_{lq}[m] \right\} \tag{28}$$

Multiuser detection systems in accord with the foregoing embodiments can be implemented in any variety of general or special purpose hardware and/or software devices. Figure 11 depicts one such implementation. In this embodiment, each frame of data is processed three times by the MUD processing card 118 (or, "MUD processor" for short), although it can be recognized that multiple such cards could be employed instead (or in addition) for this purpose. During the first pass, only the control channels are respread which the maximum ratio combination (MRC) and MUD processing is performed on the data channels. During subsequent passes, data channels are processed exclusively, with new y (i.e., soft decisions) and b (i.e., hard decisions) data being generated as shown in the diagram.

Amplitude ratios and amplitudes are determined via the DSP (e.g., element 900, or a DSP otherwise coupled with the processor board 118 and receiver 110), as well as certain

waveform statistics. These values (e.g., matrices and vectors) are used by the MUD processor in various ways. The MUD processor is decomposed into four stages that closely match the structure of the software simulation: Alpha Calculation and Respread 1302, raised-cosine filtering 1304, de-spreading 1306, and MRC 1308. Each pass through the MUD processor is equivalent to one processing stage of the implementations discussed above. The design is pipelined and "parallelized." In the illustrated embodiment, the clock speed can be 132 MHz resulting in a throughput of 2.33 ms/frame, however, the clock rate and throughput varies depending on the requirements. The illustrated embodiment allows for three-pass MUD processing with additional overhead from external processing, resulting in a 4-times real-time processing throughput.

The alpha calculation and respread operations 1302 are carried out by a set of thirty-two processing elements arranged in parallel. These can be processing elements within an ASIC, FPGA, PLD or other such device, for example. Each processing element processes two users of four fingers each. Values for b are stored in a double-buffered lookup table. Values of â and $j\hat{a}$ are pre-multiplied with beta by an external processor and stored in a quad-buffered lookup table. The alpha calculation state generated the following values for each finger, where subscripts indicate antenna identifier:

$$\alpha_0 = \beta_0 \cdot (C \cdot \hat{a}_0 - jC \cdot j\hat{a}_0)$$

$$j\alpha_0 = \beta_0 \cdot (jC \cdot \hat{a}_0 + C \cdot j\hat{a}_0)$$

$$\alpha 1 = \beta_1 \cdot (C \cdot \hat{a}_1 - jC \cdot j\hat{a}_1)$$

$$j\alpha_1 = \beta_1 \cdot (jC \cdot \hat{a}_1 + C \cdot j\hat{a}_1)$$

These values are accumulated during the serial processing cycle into four independent 8-times oversampling buffers. There are eight memory elements in each buffer and the element used is determined by the sub-chip delay setting for each finger.

Once eight fingers have been accumulated into the oversampling buffer, the data is passed into set of four independent adder-trees. These adder-trees each termination in a single output, completing the respread operation. The four raised-cosine filters 1304 convolve the alpha data with a set of weights determined by the following equation:

$$g_{rc}(t)=\frac{\sin\left(\pi\frac{1}{t}\right)\cdot\cos\left(\alpha\pi\frac{1}{T}\right)}{\pi\frac{1}{t}\left(1-\left(2\alpha\frac{1}{T}\right)^2\right)}$$

The filters can be implemented with 97 taps with odd symmetry. The filters illustrated run at 8-times the chip rate, however, other rates are possible. The filters can be implemented in a variety of compute elements 220, or other devices such as ASICs, FPGAs for example.

The despread function 1306 can be performed by a set of thirty-two processing elements arranged in parallel. Each processing element serially processes two users of four fingers each. For each finger, one chip value out of eight, selected based on the sub-chip delay, is accepted from the output of the raised-cosine filter. The despread state performs the following calculations for each finger (subscripts indicate antenna):

$$y_0=\sum_{0}^{SF-1}C\cdot r_0+jC\cdot jr_0$$

$$jy_0=\sum_{0}^{SF-1}C\cdot jr_0-jC\cdot r_0$$

$$y_1=\sum_{0}^{SF-1}C\cdot r_1+jC\cdot jr_1$$

$$jy_1=\sum_{0}^{SF-1}C\cdot jr_1-jC\cdot r_1$$

The MRC operations are carried out by a set of four processing elements arranged in parallel, such as the compute elements 220 for example. Each processor is capable of serially processing eight users of four fingers each. Values for y are stored in a double-buffered lookup table. Values for b are derived from the MSB of the y data. Note that the b data used in the MUD stage is independent of the b data used in the respread stage. Values of $\hat{a}$ and $j\hat{a}$ <are pre-multiplied with β by an external processor and stored in a quad-buffered lookup table. Also, $\sum(\hat{a}^2+j\hat{a}^2)$ for each channel is stored in a quad-buffered table.

The output stage contains a set of sequential destination buffer pointers for each channel. The data generated by each channel, on a slot basis, is transferred to the crossbar (or other interconnect) destination indicated by these buffers. The first word of each of these transfers will contain a counter in the lower sixteen bits indicating how many y values were generated.

The upper sixteen bits will contain the constant value 0xAA55. This will allow the DSP to avoid interrupts by scanning the first word of each buffer. In addition, the DSP_UPDATE register contains a pointer to single crossbar location. Each time a slot or channel data is transmitted, an internal counter is written to this location. The counter is limited to 10 bits and will wrap around with a terminal count value of 1023.

The method of operation for the long-code multiple user detection algorithm (LCMUD) is as follows. Spread factor for four-channels requires significant amount of data transfer. In order to limit the gate count of the hardware implementation, processing an SF4 channel can result in reduced capability.

A SF4 user can be processed on certain hardware channels. When one of these special channels is operating on an SF4 user, the next three channels are disabled and are therefore unavailable for processing. This relationship is as shown in the following table:

SF4 Chan	Disabled Channels	SF4 Chan	Disabled Channels
0	1, 2, 3	32	33, 34, 35
4	5, 6, 7	36	37, 38, 39
8	9, 10, 11	40	41, 42, 43
12	12, 14, 15	44	45, 46, 47
16	17, 18, 19	48	49, 50, 51
20	21, 22, 23	52	53, 54, 55
24	25, 26, 27	56	57, 58, 59
28	29, 30, 31	60	61, 62, 63

The default y and b data buffers do not contain enough space for SF4 data. When a channel is operating on SF4 data, the y and b buffers extend into the space of the next channel in sequence. For example, if channel 0 is processing SF data, the channel 0 and channel 1 b buffers are merged into a single large buffer of 0x40 32-bit words. The y buffers are merged similarly.

In typical operation, the first pass of the LCMUD algorithm will respread the control channels in order to remove control interference. For this pass, the b data for the control channels should be loaded into BLUT while the y data for data channels should be loaded into YDEC. Each channel should be configured to operate at the spread factor of the data channel stored into the YDEC table.

Control channels are always operated at SF 256, so it is likely that the control data will need to be replicated to match the data channel spread factor. For example, each bit (b entry)

of control data would be replicated 64 times if that control channel were associated with an SF 4 data channel.

Each finger in a channel arrives at the receiver with a different delay. During the Respread operation, this skew among the fingers is recreated. During the MRC stage of MUD processing, it is necessary to remove this skew and realign the fingers of each channel. This is accomplished in the MUD processor by determining the first bit available from the most delayed finger and discarding all previous bits from all other fingers. The number of bits to discard can be individually programmed for each finger with the Discard field of the MUD-PARAM registers. This operation will typically result in a 'short' first slot of data. This is unavoidable when the MUD processor is first initialized and should not create any significant problems. The entire first slot of data can be completely discarded if 'short' slots are undesirable.

A similar situation will arise each time processing is begun on a frame of data. To avoid losing data, it is recommended that a partial slot of data from the previous frame be overlapped with the new frame. Trimming any redundant bits created this way can be accomplished with the Discard register setting or in the system DSP. In order to limit memory requirements, the LCMUD FPGA processes one slot of data at a time. Doubling buffering is used for b and y data so that processing can continue as data is streamed in. Filling these buffers is complicated by the skew that exists among fingers in a channel.

Figure 12 illustrates the skew relationship among fingers in a channel and among the channels themselves. The illustrated embodiment allows for 20us (77.8 chips) of skew among fingers in a channel and certain skew among channels, however, in other embodiments these skew allowances vary.

There are three related problems that are introduced by skew: Identifying frame & slot boundaries, populating b and y tables and changing channel constants. Because every finger of every channel can arrive at a different time, there are no universal frame and slot boundaries. The DSP must select an arbitrary reference point. The data stored in b & y tables is likely to come from two adjacent slots.

Because skew exists among fingers in a channel, it is not enough to populate the b & y tables with 2,560 sequential chips of data. There must be some data overlap between buffers to allow lagging channels to access "old" data. The amount of overlap can be calculated dynamically or fixed at some number greater than 78 and divisible by four (e.g. 80 chips). The starting point for each register is determined by the Chip Advance field of the MUDPARAM register.

A related problem is created by the significant skew among channels. As can be seen in Figure 12, Channel 0 is receiving Slot 0 while Channel 1 is receiving Slot 2. The DSP must take this skew into account when generating the b and y tables and temporally align channel data.

Selecting an arbitrary "slot" of data from a channel implies that channel constants tied to the physical slot boundaries may change while processing the arbitrary slot. The Constant Advance field of the MUDPARAM register is used to indicate when these constants should change. Registers affected this way are quad-buffered. Before data processing begins, at least two of these buffers should be initialized. During normal operation, one additional buffer is initialized for each slot processed. This system guarantees that valid constants data will always be available.

The following two tables shown the long-code MUD FPGA memory map and control/ status register:

Start Addr	End Addr	Name	Description
0000_0000	0000_0000	CSR	Control & Status Register
0000_0008	0000_000C	DSP_UPDATE	Route & Address for DSP updating
0001_0000	0001_FFFF	MUDPARAM	MUD Parameters
0002_0000	0002_FFFF	CODE	Spreading Codes
0003_0000	0004_FFFF	BLUT	Respread: b Lookup Table
0005_0000	0005_FFFF	BETA_A	Respread: Beta * a_hat Lookup Table
0006_0000	0007_FFFF	YDEC	MUD & MRC: y Lookup Table
0008_0000	0008_FFFF	ASQ	MUD & MRC: Sum a_hat squared LUT
000A_0000	000A_FFFF	OUTPUT	Output Routes & Addresses

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Name	Reserved															
R/W	RO															
Reset	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Reserved							YB	CBUF		A1	A0	R1	R0	Lst	Rst
R/W	RO							RO	RO		RO	RO	Rw	Rw	Rw	Rw
Reset	X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0

The register YB indicates which of two y and b buffers are in use. If the system is currently not processing, YB indicates the buffer that will be used when processing is initiated.

CBUF indicates which of four round-robin buffers for MUD constants (a^ beta) is currently in use. Finger skew will result in some fingers using a buffer one in advance of this indicator. To guarantee that valid data is always available, two full buffers should be initialized before operation begins. If the system is currently not processing, CBUF indicates the buffer that will be used when processing is restarted. It is technically possible to indicate precisely which buffer is in use for each finger in both the Respread and Despread processing stages. However, this would require thirty-two 32-bit registers. Implementing these registers would be costly, and the information is of little value.

A1 and A0 indicate which y and b buffers are currently being processed. A1 and A0 will never indicate '1' at the same time. An indication of '0' for both A1 and A0 means that MUD processor is idle. R1 and R0 are writable fields that indicate to the MUD processor that data is available. R1 corresponds to y and b buffer 1 and R0 corresponds to y and b buffer 0. Writing a '1' into the correct register will initiate MUD processing. Note that these buffers follow strict round-robin ordering. The YB register indicates which buffer should be activated next.

These registers will be automatically reset to '0' by the MUD hardware once processing is completed. It is not possible for the external processor to force a '0' into these registers. A '1' in this bit indicates that this is the last slot of data in a frame. Once all available data for the slot has been processed, the output buffers will be flushed. A '1' in this bit will place the MUD processor into a reset state. The external processor must manually bring the MUD processor out of reset by writing a '0' into this bit.

DSP_UPDATE is arranged as two 32-bit registers. A RACEway™ route to the MUD DSP is stored at address 0x0000_0008. A pointer to a status memory buffer is located at address 0x0000_000C. Each time the MUD processor writes a slot of channel data to a completion buffer, an incrementing count value is written to this address. The counter is fixed at 10 bits and will wrap around after a terminal count of 1023.

A quad-buffered version of the MUD parameter control register exists for each finger to be processed. Execution begins with buffer 0 and continues in round-robin fashion. These buffers are used in synchronization with the MUD constants (Beta * a_hat, etc.) buffers. Each finger is provided with an independent register to allow independent switching of constant values at slot and frame boundaries. The following table shows offsets for each MUD channel:

Offset	User		Offset	User		Offset	User		Offset	User
0x0000	0		0x0400	16		0x0800	32		0x0C00	48
0x0040	1		0x0440	17		0x0840	33		0x0C40	49
0x0080	2		0x0480	18		0x0880	34		0x0C80	50
0x00C0	3		0x04C0	19		0x08C0	35		0x0CC0	51
0x0100	4		0x0500	20		0x0900	36		0x0D00	52
0x0140	5		0x0540	21		0x0940	37		0x0D40	53
0x0180	6		0x0580	22		0x0980	38		0x0D80	54
0x01C0	7		0x05C0	23		0x09C0	39		0x0DC0	55
0x0200	8		0x0600	24		0x0A00	40		0x0E00	56
0x0240	9		0x0640	25		0x0A40	41		0x0E40	57
0x0280	10		0x0680	26		0x0A80	42		0x0E80	58
0x02C0	11		0x06C0	27		0x0AC0	43		0x0EC0	59
0x0300	12		0x0700	28		0x0B00	44		0x0F00	60
0x0340	13		0x0740	29		0x0B40	45		0x0F40	61
0x0380	14		0x0780	30		0x0B80	46		0x0F80	62
0x03C0	15		0x07C0	31		0x0BC0	47		0x0FC0	63

The following table shows buffer offsets within each channel:

Offset	Finger	Buffer
0x0000	0	0
0x0004		1
0x0008		2
0x000C		3
0x0010	1	0
0x0014		1
0x0018		2
0x001C		3
0x0020	2	0
0x0024		1
0x0028		2
0x002C		3
0x0030	3	0
0x0034		1
0x0038		2
0x003C		3

The following table shown details of the control register:

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Name	Spread Factor			Subchip Delay			Discard									
R/W	RW			RW			RW									
Reset	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Chip Advance								Constant Advance							
R/W	RW								RW							
Reset	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

The spread factor field determines how many chip samples are used to generate a data bit. In the illustrated embodiment, all fingers in a channel have the same spread factor setting, however, it can be appreciated by one skilled in the art that such constant factor setting can be variable in other embodiments. The spread factor is encoded into a 3-bit value as shown in the following table:

SF Factor	Spread Factor
000	256
001	128
010	64
011	32
100	16
101	8
110	4
111	RESERVED

The field specifies the sub-chip delay for the finger. It is used to select one of eight accumulation buffers prior to summing all Alpha values and passing them into the raised-cosine filter. Discard determines how many MUD-processed soft decisions (*y* values) to discard at the start of processing. This is done so that the first y value from each finger corresponds to the same bit. After the first slot of data is processed, the Discard field should be set to zero.

The behavior of the discard field is different than that of other register fields. Once a non-zero discard setting is detected, any new discard settings from switching to a new table entry are ignored until the current discard count reaches zero. After the count reaches zero, a new discard setting may be loaded the next time a new table entry is accessed.

All fingers within a channel will arrive at the receiver with different delays. Chip Advance is used to recreate this signal skew during the Respread operation. Y and b buffers are arranged with older data occupying lower memory addresses. Therefore, the finger with the earliest arrival time has the highest value of chip advance. Chip Advanced need not be a multiple of Spread Factor.

Constant advance indicates on which chip this finger should switch to a new set of constants (e.g. aˆ) and a new control register setting. Note that the new values take effect on the chip after the value stored here. For example, a value of 0x0 would cause the new constants to take effect on chip 1. A value of 0xFF would cause the new constants to take effect on chip 0 of the next slot. The b lookup tables are arranged as shown in the following table. B values each occupy two bits of memory, although only the LSB is utilized by LCMUD hardware.

Offset	Buffer	Offset	Buffer	Offset	Buffer	Offset	Buffer
0x0000	U0 B0	0x0400	U16 B0	0x0800	U32 B0	0x0C00	U48 B0
0x0020	U1 B0	0x0420	U17 B0	0x0820	U33 B0	0x0C20	U49 B0
0x0040	U0 B1	0x0440	U16 B1	0x0840	U32 B1	0x0C40	U48 B1
0x0060	U1 B1	0x0460	U17 B1	0x0860	U33 B1	0x0C60	U49 B1
0x0080	U2 B0	0x0480	U18 B0	0x0880	U34 B0	0x0C80	U50 B0
0x00A0	U3 B0	0x04A0	U19 B0	0x08A0	U35 B0	0x0CA0	U51 B0
0x00C0	U2 B1	0x04C0	U18 B1	0x08C0	U34 B1	0x0CC0	U50 B1
0x00E0	U3 B1	0x04E0	U19 B1	0x08E0	U35 B1	0x0CE0	U51 B1
0x0100	U4 B0	0x0500	U20 B0	0x0900	U36 B0	0x0D00	U52 B0
0x0120	U5 B0	0x0520	U21 B0	0x0920	U37 B0	0x0D20	U53 B0
0x0140	U4 B1	0x0540	U20 B1	0x0940	U36 B1	0x0D40	U52 B1
0x0160	U5 B1	0x0560	U21 B1	0x0960	U37 B1	0x0D60	U53 B1
0x0180	U6 B0	0x0580	U22 B0	0x0980	U38 B0	0x0D80	U54 B0
0x01A0	U7 B0	0x05A0	U23 B0	0x09A0	U39 B0	0x0DA0	U55 B0
0x01C0	U6 B1	0x05C0	U22 B1	0x09C0	U38 B1	0x0DC0	U54 B1
0x01E0	U7 B1	0x05E0	U23 B1	0x09E0	U39 B1	0x0DE0	U55 B1
0x0200	U8 B0	0x0600	U24 B0	0x0A00	U40 B0	0x0E00	U56 B0
0x0220	U9 B0	0x0620	U25 B0	0x0A20	U41 B0	0x0E20	U57 B0
0x0240	U8 B1	0x0640	U24 B1	0x0A40	U40 B1	0x0E40	U56 B1
0x0260	U9 B1	0x0660	U25 B1	0x0A60	U41 B1	0x0E60	U57 B1
0x0280	U10 B0	0x0680	U26 B0	0x0A80	U42 B0	0x0E80	U58 B0
0x02A0	U11 B0	0x06A0	U27 B0	0x0AA0	U43 B0	0x0EA0	U59 B0
0x02C0	U10 B1	0x06C0	U26 B1	0x0AC0	U42 B1	0x0EC0	U58 B1
0x02E0	U11 B1	0x06E0	U27 B1	0x0AE0	U43 B1	0x0EE0	U59 B1
0x0300	U12 B0	0x0700	U28 B0	0x0B00	U44 B0	0x0F00	U60 B0
0x0320	U13 B0	0x0720	U29 B0	0x0B20	U45 B0	0x0F20	U61 B0
0x0340	U12 B1	0x0740	U28 B1	0x0B40	U44 B1	0x0F40	U60 B1
0x0360	U13 B1	0x0760	U29 B1	0x0B60	U45 B1	0x0F60	U61 B1
0x0380	U14 B0	0x0780	U30 B0	0x0B80	U46 B0	0x0F80	U62 B0
0x03A0	U15 B0	0x07A0	U31 B0	0x0BA0	U47 B0	0x0FA0	U63 B0
0x03C0	U14 B1	0x07C0	U30 B1	0x0BC0	U46 B1	0x0FC0	U62 B1
0x03E0	U15 B1	0x07E0	U31 B1	0x0BE0	U47 B1	0x0FE0	U63 B1

The following table illustrates how the two-bit values are packed into 32-bit words. Spread Factor 4 channels require more storage space than is available in a single channel buffer. To allow for SF4 processing, the buffers for an even channel and the next highest odd channel are joined together. The even channel performs the processing while the odd channel is disabled.

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Name	b(0)		b(1)		b(2)		b(3)		b(4)		b(5)		b(6)		b(7)	
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	b(8)		b(9)		b(10)		b(11)		b(12)		b(13)		b(14)		b(15)	

The beta*a-hat table contains the amplitude estimates for each finger pre-multiplied by the value of Beta. The following table shows the memory mappings for each channel.

Offset	User	Offset	User	Offset	User	Offset	User
0x0000	0	0x0800	16	0x1000	32	0x1800	48
0x0080	1	0x0880	17	0x1080	33	0x1880	49
0x0100	2	0x0900	18	0x1100	34	0x1900	50
0x0180	3	0x0980	19	0x1180	35	0x1980	51
0x0200	4	0x0A00	20	0x1200	36	0x1A00	52
0x0280	5	0x0A80	21	0x1280	37	0x1A80	53
0x0300	6	0x0B00	22	0x1300	38	0x1B00	54
0x0380	7	0x0B80	23	0x1380	39	0x1B80	55
0x0400	8	0x0C00	24	0x1400	40	0x1C00	56
0x0480	9	0x0C80	25	0x1480	41	0x1C80	57
0x0500	10	0x0D00	26	0x1500	42	0x1D00	58
0x0580	11	0x0D80	27	0x1580	43	0x1D80	59
0x0600	12	0x0E00	28	0x1600	44	0x1E00	60
0x0680	13	0x0E80	29	0x1680	45	0x1E80	61
0x0700	14	0x0F00	30	0x1700	46	0x1F00	62
0x0780	15	0x0F80	31	0x1780	47	0x1F80	63

The following table shows buffers that are distributed for each channel:

Offset	User Buffer
0x00	0
0x20	1
0x40	2
0x80	3

The following table shows a memory mapping for individual fingers of each antenna.

Offset	Finger	Antenna
0x00	0	0
0x04	1	
0x08	2	
0x0C	3	
0x10	0	1
0x14	1	
0x18	2	
0x1C	3	

The y (soft decisions) table contains two buffers for each channel. Like the *b* lookup table, an even and odd channel are bonded together to process SF4. Each y data value is stored as a byte. The data is written into the buffers as packed 32-bit words.

Offset	Buffer	Offset	Buffer	Offset	Buffer	Offset	Buffer
0x0000	U0 B0	0x4000	U16 B0	0x8000	U32 B0	0xC000	U48 B0
0x0200	U1 B0	0x4200	U17 B0	0x8200	U33 B0	0xC200	U49 B0
0x0400	U2 B1	0x4400	U18 B1	0x8400	U34 B1	0xC400	U50 B1
0x0600	U3 B1	0x4600	U19 B1	0x8600	U35 B1	0xC600	U51 B1
0x0800	U0 B0	0x4800	U16 B0	0x8800	U32 B0	0xC800	U48 B0
0x0A00	U1 B0	0x4A00	U17 B0	0x8A00	U33 B0	0xCA00	U49 B0
0x0C00	U2 B1	0x4C00	U18 B1	0x8C00	U34 B1	0xCC00	U50 B1
0x0E00	U3 B1	0x4E00	U19 B1	0x8E00	U35 B1	0xCE00	U51 B1
0x0000	U4 B0	0x5000	U20 B0	0x9000	U36 B0	0xD000	U52 B0
0x0200	U5 B0	0x5200	U21 B0	0x9200	U37 B0	0xD200	U53 B0
0x0400	U6 B1	0x5400	U22 B1	0x9400	U38 B1	0xD400	U54 B1
0x0600	U7 B1	0x5600	U23 B1	0x9600	U39 B1	0xD600	U55 B1
0x0800	U4 B0	0x5800	U20 B0	0x9800	U36 B0	0xD800	U52 B0
0x0A00	U5 B0	0x5A00	U21 B0	0x9A00	U37 B0	0xDA00	U53 B0
0x0C00	U6 B1	0x5C00	U22 B1	0x9C00	U38 B1	0xDC00	U54 B1
0x0E00	U7 B1	0x5E00	U23 B1	0x9E00	U39 B1	0xDE00	U55 B1
0x0000	U8 B0	0x6000	U24 B0	0xA000	U40 B0	0xE000	U56 B0
0x0200	U9 B0	0x6200	U25 B0	0xA200	U41 B0	0xE200	U57 B0
0x0400	U10 B1	0x6400	U26 B1	0xA400	U42 B1	0xE400	U58 B1
0x0600	U11 B1	0x6600	U27 B1	0xA600	U43 B1	0xE600	U59 B1
0x0800	U8 B0	0x6800	U24 B0	0xA800	U40 B0	0xE800	U56 B0
0x0A00	U9 B0	0x6A00	U25 B0	0xAA00	U41 B0	0xEA00	U57 B0
0x0C00	U10 B1	0x6C00	U26 B1	0xAC00	U42 B1	0xEC00	U58 B1
0x0E00	U11 B1	0x6E00	U27 B1	0xAE00	U43 B1	0xEE00	U59 B1
0x0000	U12 B0	0x7000	U28 B0	0xB000	U44 B0	0xF000	U60 B0
0x0200	U13 B0	0x7200	U29 B0	0xB200	U45 B0	0xF200	U61 B0
0x0400	U14 B1	0x7400	U30 B1	0xB400	U46 B1	0xF400	U62 B1
0x0600	U15 B1	0x7600	U31 B1	0xB600	U47 B1	0xF600	U63 B1
0x0800	U12 B0	0x7800	U28 B0	0xB800	U44 B0	0xF800	U60 B0
0x0A00	U13 B0	0x7A00	U29 B0	0xBA00	U45 B0	0xFA00	U61 B0
0x0C00	U14 B1	0x7C00	U30 B1	0xBC00	U46 B1	0xFC00	U62 B1
0x0E00	U15 B1	0x7E00	U31 B1	0xBE00	U47 B1	0xFE00	U63 B1

The sum of the a-hat squares is stored as a 16-bit value. The following table contains a memory address mapping for each channel.

0x0000	0	0x0200	16	0x0400	32	0x0600	48
Offset	User	Offset	User	Offset	User	Offset	User
0x0020	1	0x0220	17	0x0420	33	0x0620	49
0x0040	2	0x0240	18	0x0440	34	0x0640	50
0x0060	3	0x0260	19	0x0460	35	0x0660	51
0x0080	4	0x0280	20	0x0480	36	0x0680	52
0x00A0	5	0x02A0	21	0x04A0	37	0x06A0	53
0x00C0	6	0x02C0	22	0x04C0	38	0x06C0	54
0x00E0	7	0x02E0	23	0x04E0	39	0x06E0	55
0x0100	8	0x0300	24	0x0500	40	0x0700	56
0x0120	9	0x0320	25	0x0520	41	0x0720	57
0x0140	10	0x0340	26	0x0540	42	0x0740	58
0x0160	11	0x0360	27	0x0560	43	0x0760	59
0x0180	12	0x0380	28	0x0580	44	0x0780	60
0x01A0	13	0x03A0	29	0x05A0	45	0x07A0	61
0x01C0	14	0x03C0	30	0x05C0	46	0x07C0	62
0x01E0	15	0x03E0	31	0x05E0	47	0x07E0	63

Within each buffer, the value for antenna 0 is stored at address offset 0x0 with the value for antenna one stored at address offset 0x04. The following table demonstrates a mapping for each finger.

Offset	User Buffer
0x00	0
0x08	1
0x10	2
0x1C	3

Each channel is provided a crossbar (e.g., RACEway™) route on the bus, and a base address for buffering output on a slot basis. Registers for controlling buffers are allocated as shown in the following two tables. External devices are blocked from writing to register addresses marked as reserved.

Offset	User	Offset	User	Offset	User	Offset	User
0x0000	0	0x0200	16	0x0400	32	0x0600	48
0x0020	1	0x0220	17	0x0420	33	0x0620	49
0x0040	2	0x0240	18	0x0440	34	0x0640	50
0x0060	3	0x0260	19	0x0460	35	0x0660	51

0x0080	4	0x0280	20	0x0480	36	0x0680	52
0x00A0	5	0x02A0	21	0x04A0	37	0x06A0	53
0x00C0	6	0x02C0	22	0x04C0	38	0x06C0	54
0x00E0	7	0x02E0	23	0x04E0	39	0x06E0	55
0x0100	8	0x0300	24	0x0500	40	0x0700	56
0x0120	9	0x0320	25	0x0520	41	0x0720	57
0x0140	10	0x0340	26	0x0540	42	0x0740	58
0x0160	11	0x0360	27	0x0560	43	0x0760	59
0x0180	12	0x0380	28	0x0580	44	0x0780	60
0x01A0	13	0x03A0	29	0x05A0	45	0x07A0	61
0x01C0	14	0x03C0	30	0x05C0	46	0x07C0	62
0x01E0	15	0x03E0	31	0x05E0	47	0x07E0	63

Offset	Entry
0x0000	Route to Channel Destination
0x0004	Base Address for Buffers
0x0008	Buffers
0x000C	RESERVED
0x0010	RESERVED
0x0014	RESERVED
0x0018	RESERVED
0x001C	RESERVED

Slot buffer size is automatically determined by the channel spread factor. Buffers are used in round-robin fashion and all buffers for a channel must be arranged contiguously. The buffers control register determines how many buffers are allocated for each channel. A setting of 0 indicates one available buffer, a setting of 1 indicates two available buffers, and so on.

A further understanding of the operation of the illustrated and other embodiments of the invention may be attained by reference to (i) US Provisional Application Serial No. 60/275,846 filed March 14, 2001, entitled "Improved Wireless Communications Systems and Methods"; (ii) US Provisional Application Serial No. 60/289,600 filed May 7, 2001, entitled "Improved Wireless Communications Systems and Methods Using Long-Code Multi-User Detection"' and (iii) US Provisional Application Serial Number. 60/295,060 filed June 1, 2001 entitled "Improved Wireless Communications Systems and Methods for a Communications Computer," the teachings all of which are incorporated herein by reference, and a copy of the latter of which may be filed herewith.

The above embodiments are presented for illustrative purposes only. Those skilled in the art will appreciate that various modifications can be made to these embodiments without

departing from the scope of the present invention. For example, multiple summations can be utilized by a system of the invention, and not separate summations as described herein. Moreover, by way of further non-limiting example, it will be appreciated that although the terminology used above is largely based on the UMTS CDMA protocols, that the methods and apparatus described herein are equally applicable to DS/CDMA, CDMA2000 1X, CDMA2000 1xEV-DO, and other forms of CDMA.

Therefore, in view of the foregoing, what we claim is:

Background of the Invention

The invention pertains to wireless communications and, more particularly, to communications computers. The invention has application, by way of non-limiting example, in improving the capacity of cellular phone base stations.

Code-division multiple access (CDMA) is used increasingly in wireless communications. It is a form of multiplexing communications, e.g., between cellular phones and base stations, based on distinct digital codes in the communication signals. This can be contrasted with other wireless protocols, such as frequency-division multiple access and time-division multiple access, in which multiplexing is based on the use of orthogonal frequency bands and orthogonal time-slots, respectively.

A limiting factor in CDMA communication and, particularly, in so-called direct sequence CDMA (DS-CDMA), is the interference between multiple simultaneous communications, e.g., multiple cellular phone users in the same geographic area using their phones at the same time. This is referred to as multiple access interference (MAI). It has effect of limiting the capacity of cellular phone base stations, since interference may exceed acceptable levels -- driving service quality below acceptable levels -- when there are too many users.

A technique known as multi-user detection (MUD) reduces multiple access interference and, as a consequence, increases base station capacity. MUD can reduce interference not only between multiple signals of like strength, but also that caused by users so close to the base station as to otherwise overpower signals from other users (the so-called near/far problem). MUD generally functions on the principle that signals from multiple simultaneous users can be jointly used to improve detection of the signal from any single user. Many forms of MUD are known; surveys are provided in Moshavi, "Multi-User Detection for DS-CDMA Systems," IEEE Communications Magazine (October, 1996) and Duel-Hallen et al, "Multiuser Detection for CDMA Systems," IEEE Personal Communications (April 1995). Though a promising solution to increasing the capacity of cellular phone base stations, MUD techniques are typically so computationally intensive as to limit practical application.

An object of this invention is to provide improved methods and apparatus for wireless communications. A related object is to provide such methods and apparatus for multi-user detection or interference cancellation in code-division multiple access communications.

A further object of the invention is to provide such methods and apparatus as can be cost-effectively implemented and as require minimal changes in existing wireless communications infrastructure.

A still further object of the invention is to provide methods and apparatus for executing multi-user detection and related algorithms in real-time.

A still further object of the invention is to provide such methods and apparatus as manage faults for high-availability.

Summary of the Invention

These and other objects are met by the invention which provides, in one aspect, a communications computer, referred to as the "MCW-1" (among other terms) in the materials that follow, and methods of operation thereof. An overview of that system is provided in the section entitled "Communications Computer," beginning on page 5 hereof. A more complete understanding of its implementation may be attained by reference to the other attached materials.

In view of those materials, aspects of the invention include, but are not limited to the following:

- architecture and operation of a communications computer for a wireless communications system, including a fully programmable computer inserted into base transceiver station (BTS) to support compute-intensive and/or highly data-dependent functions such as adaptive processing and interference cancellation

These and other aspects of the invention (including utilization of the aforementioned methods and aspects for other than wireless communications and/or interference cancellation) are evident in the materials that follow.

Detailed Description of the Invention

See the attached materials on pages 5 – 11 hereof, providing description and block diagram of a preferred structure and operation of a communications computer for wireless applications according to the invention.

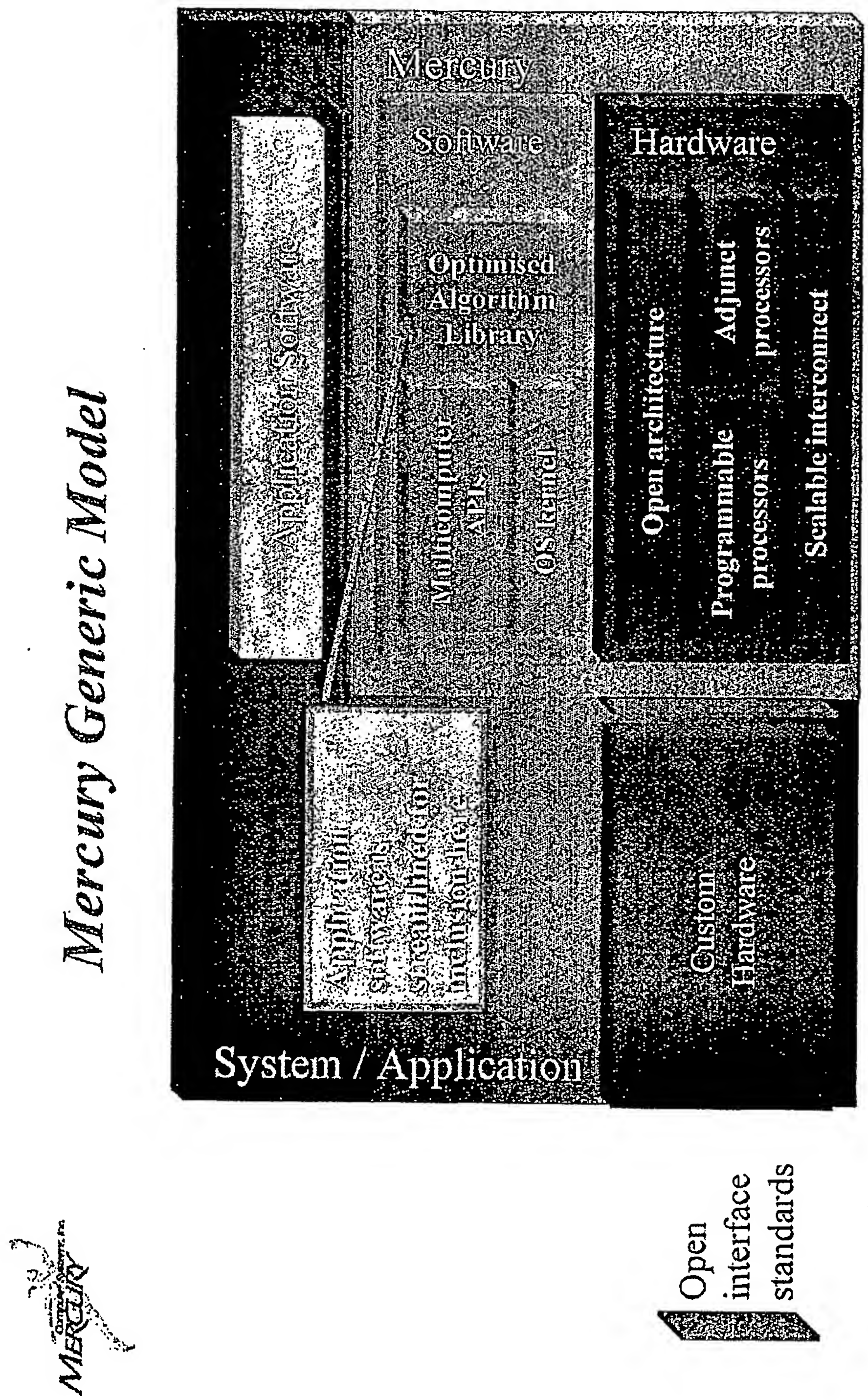
The aforementioned materials pertain to improvements on the methods and apparatus described in United States Provisional Application Serial No. 60/275,846, filed March 14, 2001, entitled IMPROVED WIRELESS COMMUNICATIONS SYSTEMS AND METHODS and United States Provisional Application Serial No. 60/289,600, filed May 7, 2001, entitled IMPROVED WIRELESS COMMUNICATIONS SYSTEMS AND METHODS USING LONG-CODE MULTI-USER DETECTION, the teachings of both of which are incorporated herein by reference and copies of at least portions of which are attached hereto. Those copies bears the U.S. Postal Service Express Mail label number of both prior filings, as well as that of this filing (the latter being referred to as the "New Exp. Mail Label No.").

Communications Computer

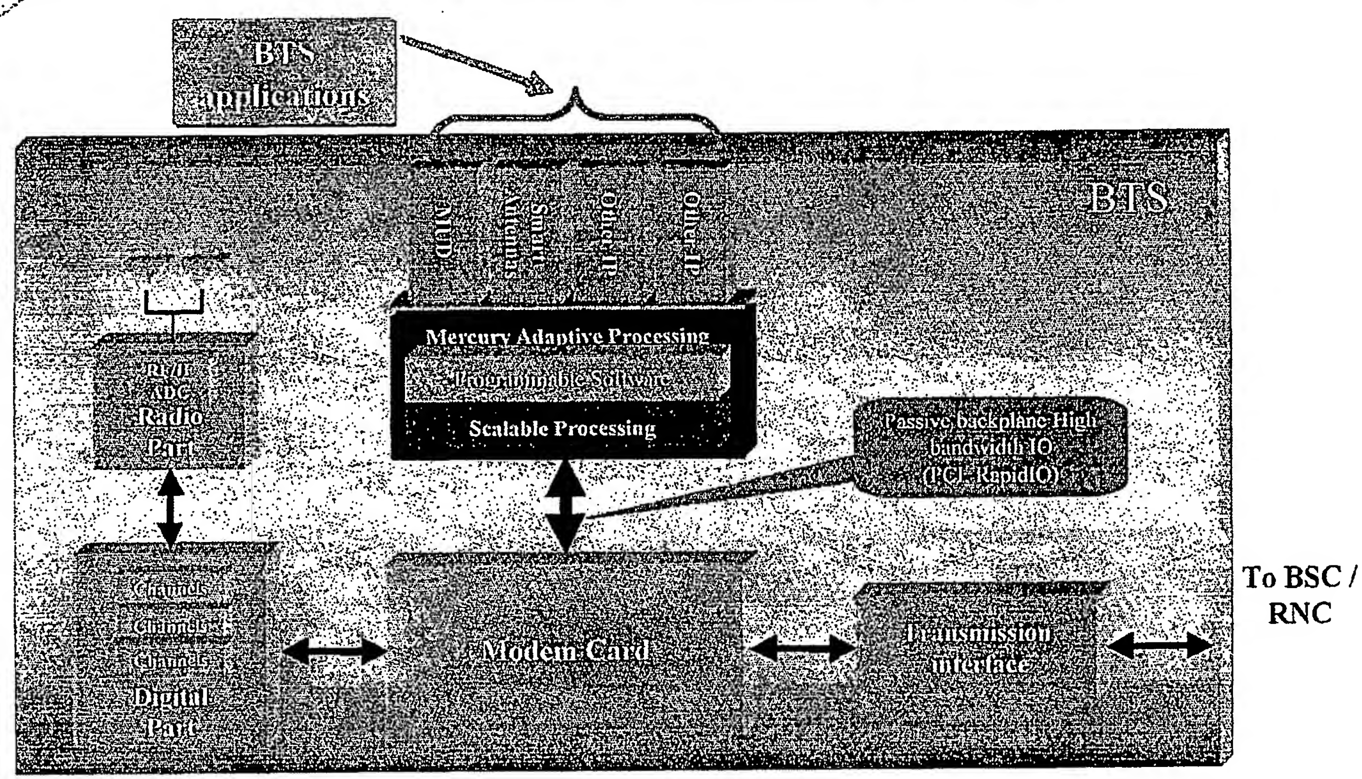
- Fully programmable computer inserted into base transceiver station (BTS) to support compute-intensive and/or highly data-dependent functions such as adaptive processing and interference cancellation
  - Overcomes rigidity of ASIC-based application implementation
  - Overcomes limitations of DSP instruction sets
  - Overcomes traditional inter-processor bandwidth limitations
    - By using modern processor interconnect technology rather than busses
  - Enables remote modification of functionality by software download
- High-profile applications:
  - Multi-user detection (MUD)
  - Interference cancellation
  - Smart and adaptive antenna processing
  - Interference avoidance

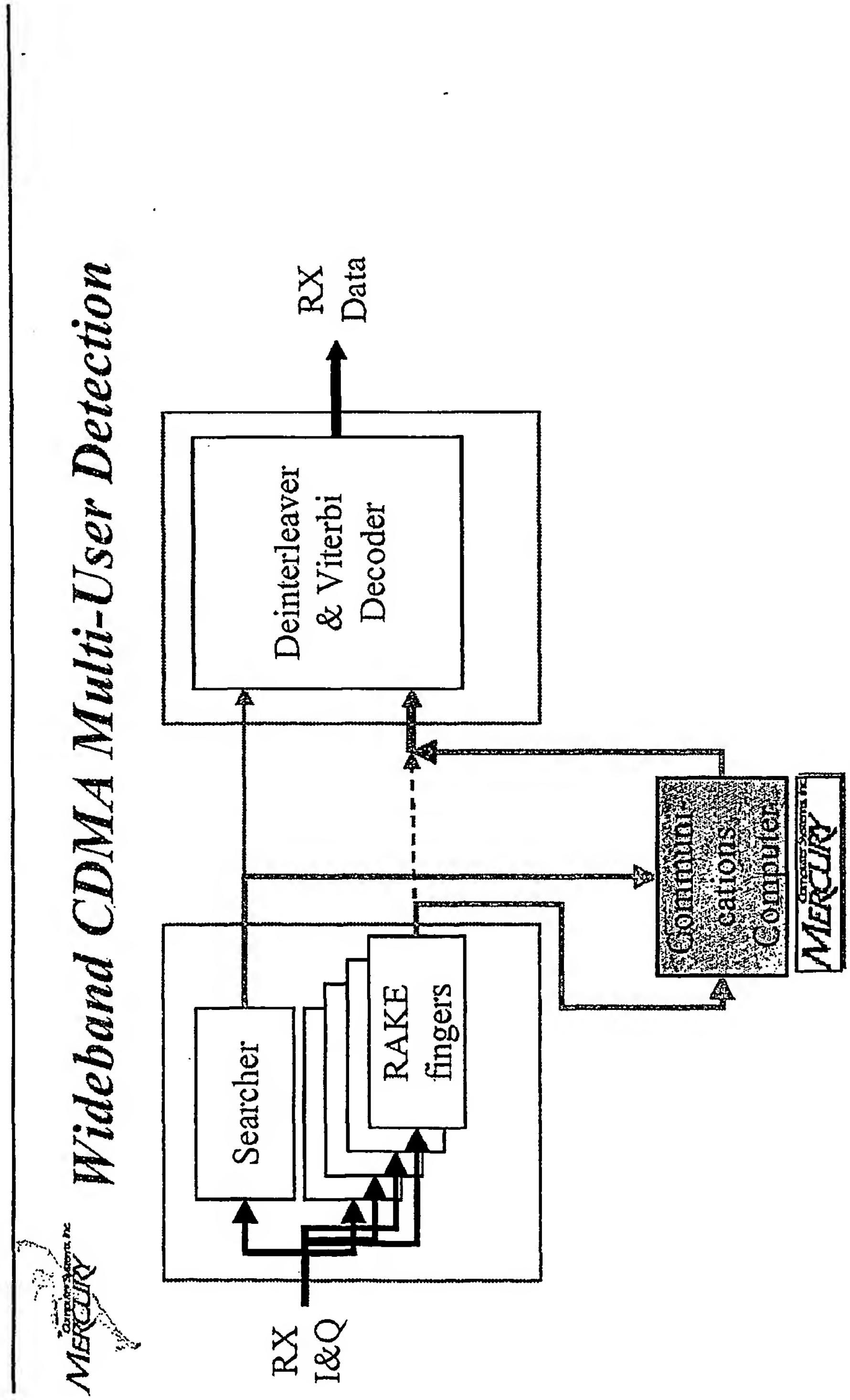
Communications Computer (Cont.)

- Multiple communications computers can be interconnected to promote
  - Load balancing among cell site sectors
  - Improved fault resilience
  - Additional algorithm sophistication/complexity
  - Functionality of additional algorithms
- Communications computer concept can be extended by interconnection to encompass full BTS functionality


Mercury Generic Model
Mercury
Software
Hardware
Optimised Algorithm Library
Multicomputer APIs
OS Kernel
Application Software
Open architecture
Programmable processors
Adjunct processors
Scalable interconnect
Custom Hardware
System / Application
Open interface standards
MERCURY

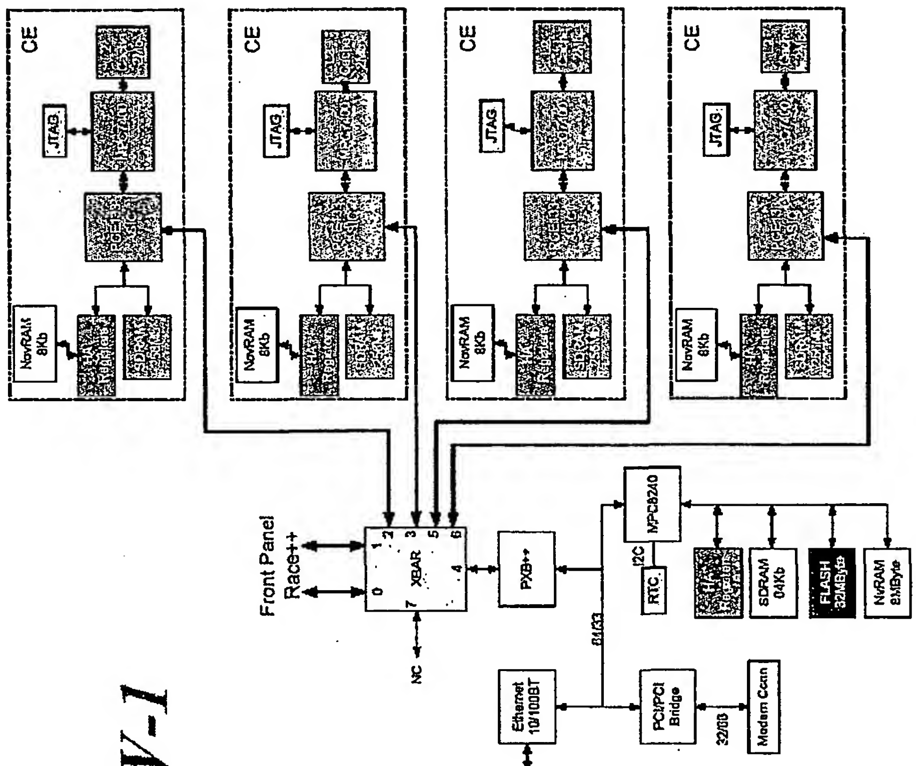
Communications Computer Concept




Wideband CDMA Multi-User Detection
RX I&Q
Searcher
RAKE fingers
Deinterleaver & Viterbi Decoder
RX Data
Communi-cations Computer
MERCURY

Communications Computer Prototype: MCW-1

- Interconnectable telco-grade multicomputer boards and system software suitable for MUD, interference cancellation, and other adaptive processing applications
  - Hardware: Four G4/Nitros and SDRAM; plus MPC8240, watchdogs, NVRAM, PCI and Enet connectivity…
    - Scalable up and down in complexity
  - Software: Application; autonomous fault monitoring, detection, isolation, recovery; automatic remote software update; remote access via embedded web server


CE
JTAG
NovRAM 8Kb
Front Panel
Race++
XBAR
NC
PXB++
MPC8240
I2C
RTC
SDRAM
FLASH 32MByte
Ethernet 10/100BT
PCI/PCI Bridge
Modem Conn

MCW-1

Background of the Invention

The invention pertains to wireless communications and, more particularly, to methods and apparatus for interference cancellation in code-division multiple access communications. The invention has application, by way of non-limiting example, in improving the capacity of cellular phone base stations.

Code-division multiple access (CDMA) is used increasingly in wireless communications. It is a form of multiplexing communications, e.g., between cellular phones and base stations, based on distinct digital codes in the communication signals. This can be contrasted with other wireless protocols, such as frequency-division multiple access and time-division multiple access, in which multiplexing is based on the use of orthogonal frequency bands and orthogonal time-slots, respectively.

A limiting factor in CDMA communication and, particularly, in so-called direct sequence CDMA (DS-CDMA), is the interference between multiple simultaneous communications, e.g., multiple cellular phone users in the same geographic area using their phones at the same time. This is referred to as multiple access interference (MAI). It has effect of limiting the capacity of cellular phone base stations, since interference may exceed acceptable levels -- driving service quality below acceptable levels -- when there are too many users.

A technique known as multi-user detection (MUD) reduces multiple access interference and, as a consequence, increases base station capacity. MUD can reduce interference not only between multiple signals of like strength, but also that caused by users so close to the base station as to otherwise overpower signals from other users (the so-called near/far problem). MUD generally functions on the principle that signals from multiple simultaneous users can be jointly used to improve detection of the signal from any single user. Many forms of MUD are known; surveys are provided in Moshavi, "Multi-User Detection for DS-CDMA Systems," IEEE Communications Magazine (October, 1996) and Duel-Hallen et al, "Multiuser Detection for CDMA Systems," IEEE Personal Communications (April 1995). Though a promising solution to increasing the capacity of cellular phone base stations, MUD techniques are typically so computationally intensive as to limit practical application.

An object of this invention is to provide improved methods and apparatus for wireless communications. A related object is to provide such methods and apparatus for multi-user detection or interference cancellation in code-division multiple access communications.

A further object of the invention is to provide such methods and apparatus as can be cost-effectively implemented and as require minimal changes in existing wireless communications infrastructure.

A still further object of the invention is to provide methods and apparatus for executing multi-user detection and related algorithms in real-time.

A still further object of the invention is to provide such methods and apparatus as manage faults for high-availability.

Summary of the Invention

These and other objects are met by the invention which provides, in one aspect, a wireless communications system, referred to as the "MCW-1" (among other terms) in the materials that follow, and methods of operation thereof. An overview of that system is provided in the document entitled "Software Architecture of the MCW-1 MUD Board," immediately following this Summary. A more complete understanding of its implementation may be attained by reference to the other attached materials.

In view of those materials, aspects of the invention include, but are not limited to, the following:

- methods and apparatus for long-code multi-user detection (MUD) in a wireless communications system.

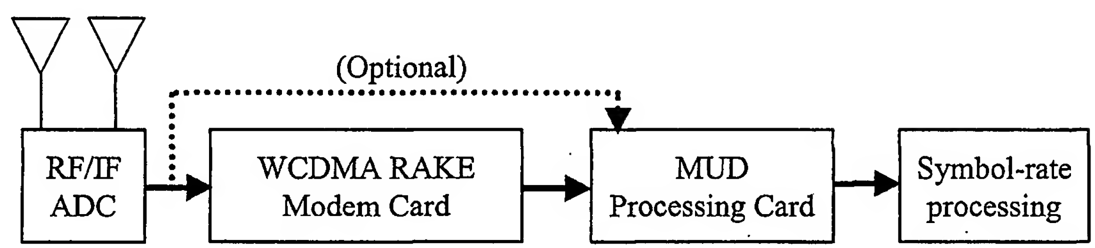
These and other aspects of the invention (including utilization of the aforementioned methods and aspects for other than wireless communications and/or interference cancellation) are evident in the materials that follow.

Detailed Description of the Invention

See the attached materials on pages 5 – 12 hereof, providing a block diagram of a preferred algorithm for long code MUD which includes identification of (roughly) how many GOPS are involved in each major function; a diagram showing interfaces between a long code MUD processing card according to the invention and a modem, e.g., of the type provided by Motorola (or another supplier of such components); and two block diagrams of the same BASELINE 0 board hardware architecture at a top level identifiying the processing nodes. The attached diagram entitled "Long-code Mapping to Hardware" illustrates support of 64 users for long code MUD and shows parts of the long code MUD algorithm supported by each processing node. The diagram entitled "Short-code Mapping to Hardware" illustrates support of 128 users for short code MUD and shows parts of the short code MUD algorithm would be supported by each processing node.

The aforementioned materials pertain to improvements on the methods and apparatus described in United States Provisional Application Serial No. 60/275,846, filed March 14, 2001, entitled IMPROVED WIRELESS COMMUNICATIONS SYSTEMS AND METHODS, the teachings of which are incorporated herein by reference and a copy of which is attached hereto. That copy bears the U.S. Postal Service Express Mail label number of both the original filing, as well as that of this filing (the latter being referred to as the "New Exp. Mail Label No.").

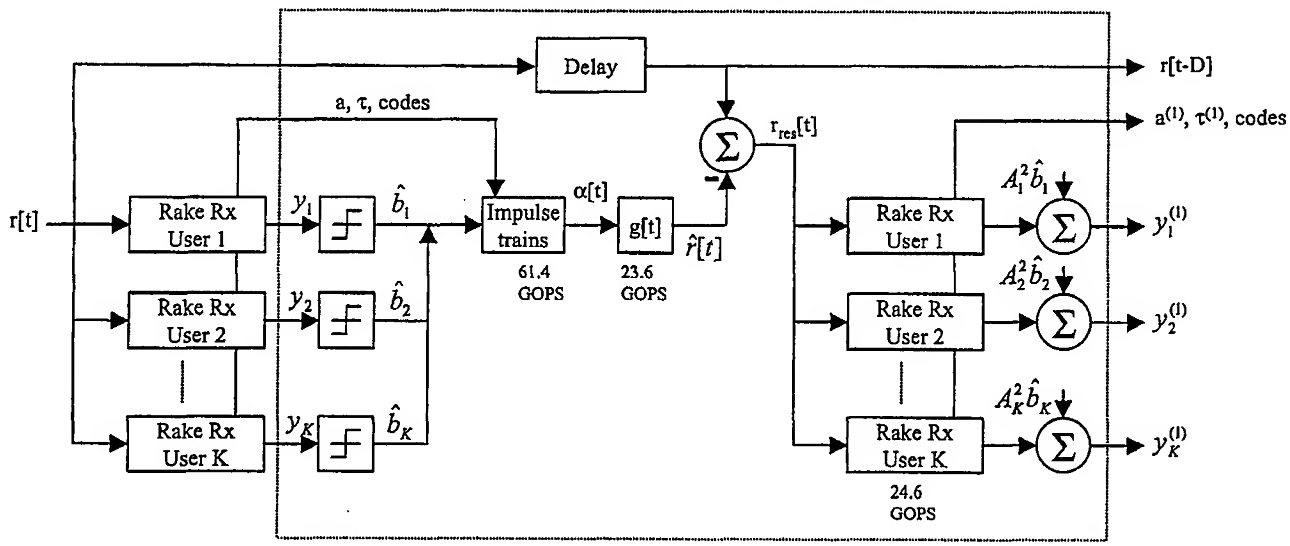
Long-Code Multiuser Detection Enhancement Concept



Optional antenna-stream input to MUD processing card allows multiple-stage interference cancellation and multiuser channel-amplitude estimation.

MERCURY Computer Systems, Inc.

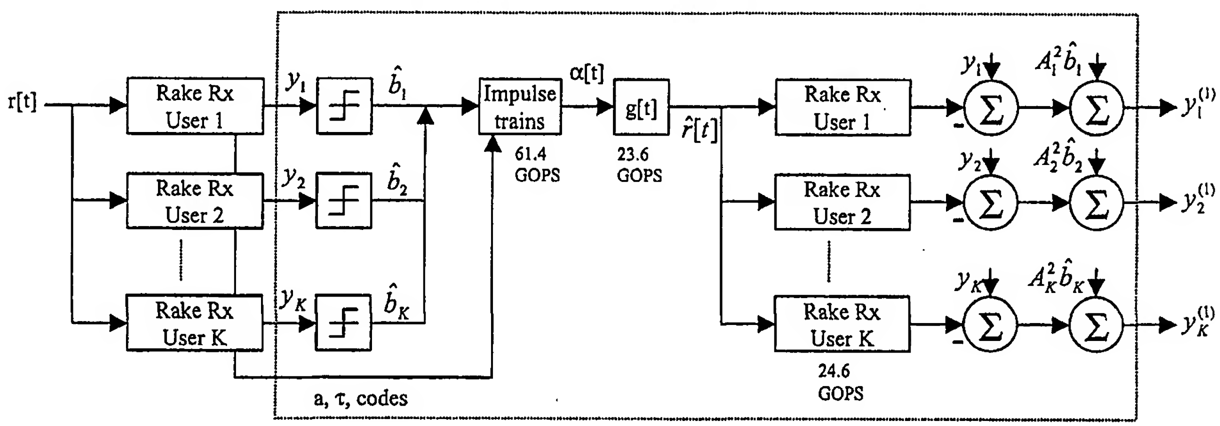
Block Diagram of Multiple-Stage Long-Code Algorithm



Computational complexity figures (GOPS) are based on
- 128 SF 256 users
- 4 multipath fingers
- 8 samples per chip

Mercury Computer Systems, Inc.

Block Diagram of Single-Stage Long-Code Algorithm

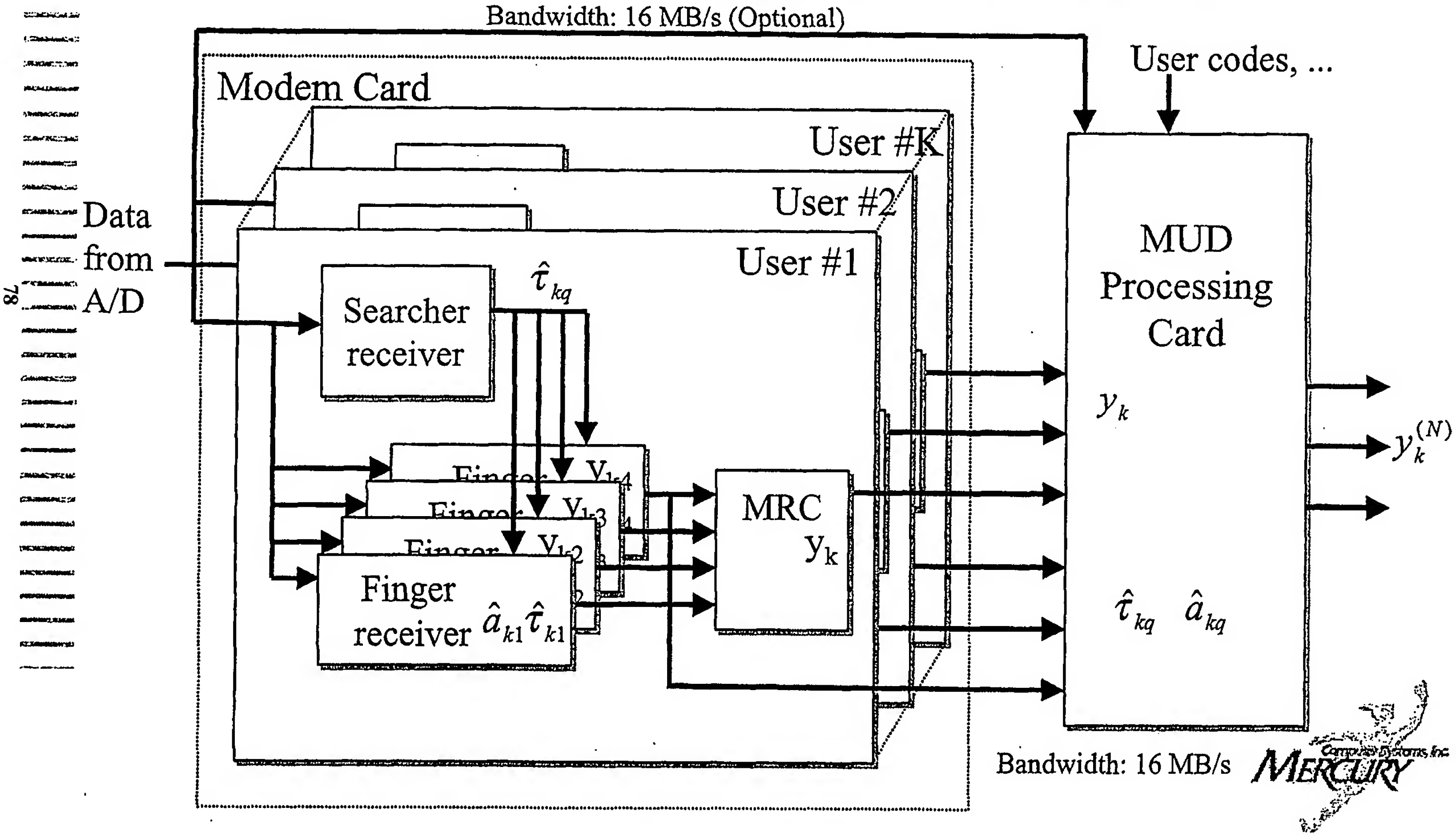


Computational complexity figures (GOPS) are based on

- 128 SF 256 users
- 4 multipath fingers
- 8 samples per chip

MERCURY Computer Systems, Inc.

MUD Interface to Modem



Data Transferred to or from MUD Processing Card

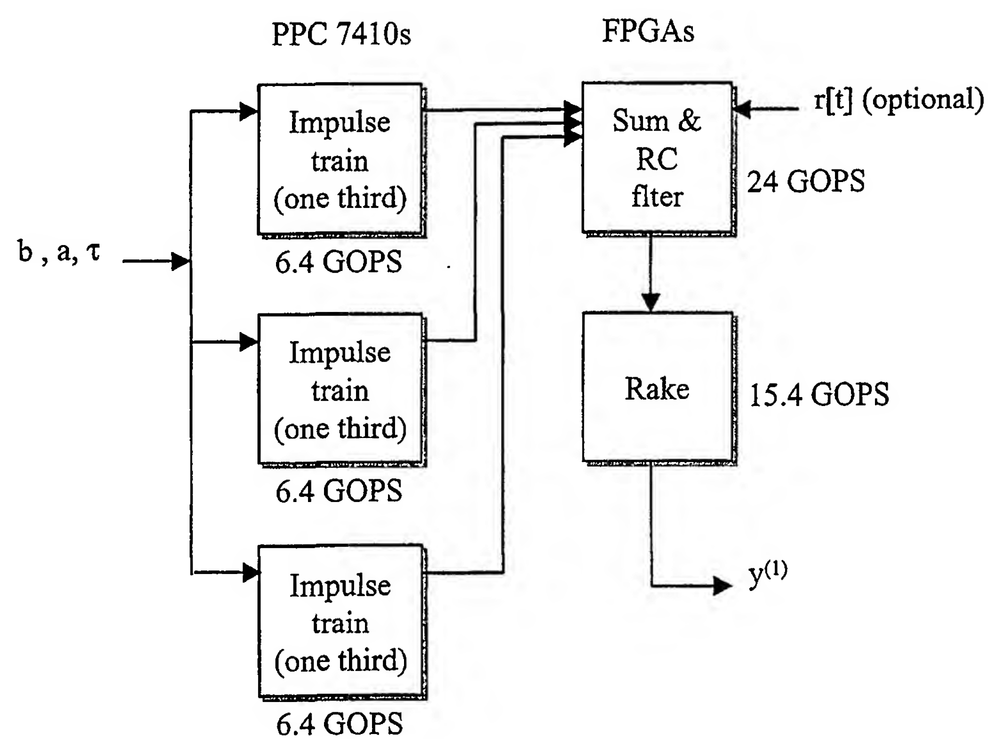
- Inputs
  - Frame number
  - Post-MRC matched-filter outputs y_k for DPCCH and DPDCHs
  - Number of DPDCHs
  - DPCCH slot format
  - Number of rake fingers
  - Number of antennas used
  - Channel amplitude estimates a_{kq}
  - Channel lag estimates τ_{kq}
  - Spreading factor SF_k
  - Code number
  - Compressed mode information
    - Compressed mode flag, Compressed mode frame, N_first, TGL
  - Amplitude ratios β_{dk}, β_{ck}
  - Antenna streams $r[t]$ (Optional)
- Outputs
  - Cleaned data bit estimates b_k

Long-code MUD Processing Card Interface Bandwidth

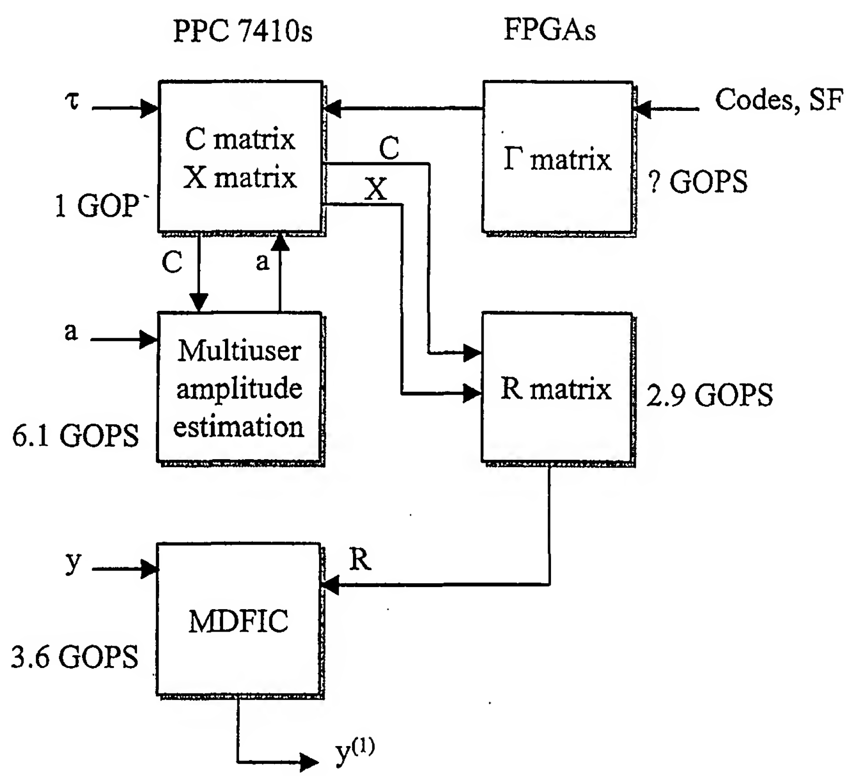
Data Description	BW (MB/s)
Antenna streams	16.00
Post-MRC outputs y_k	3.260
Channel amplitude estimates a_{kq}	6.144
Channel lag estimates τ_{kq}	0.001
Cleaned data bit estimates b_k	1.920
TOTAL	27.325

Mercury Computer Systems, Inc.

Long-code Mapping to Hardware



Short-code Mapping to Hardware



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES PROVISIONAL PATENT APPLICATION

for

IMPROVED WIRELESS COMMUNICATIONS SYSTEMS AND METHODS

Inventors:

John H. Oates
598 Seaverns Bridge Road
Amherst, New Hampshire 03031

Alden J. Fuchs
160 Pine Hill Road
Nashua, New Hampshire 03063

Jonathan E. Greene
83n Hollenbeck Avenue
Great Barrington, Massachusetts 01230

Frank P. Lauginiger
772 Russell Station Road
Francestown, New Hampshire 03043

Paul E. Cantrell
15 Prescott Drive
Chelmsford, Massachusetts 01863

Jan N. Dunn
48 North Court Street, #1
Providence, Rhode Island 02903

Steven R. Imperiali
43 Haynes Road
Townsend, Massachusetts 01469

Kathleen J. Jacques
10 Juniper Street
Jay, Maine 04239

William J. Jenkins
61 Heritage Lane #C4
Leominster, Massachusetts 01453

David E. Majchrzak
11 Regine Street
Hudson, New Hampshire 03051

Mirza Cifric
705 Mass Avenue
Boston, Massachusetts 02118

Michael J. Vinskus
5 Cranberry Lane
Litchfield, New Hampshire 03052

Background of the Invention

The invention pertains to wireless communications and, more particularly, to methods and apparatus for interference cancellation in code-division multiple access communications. The invention has application, by way of non-limiting example, in improving the capacity of cellular phone base stations.

Code-division multiple access (CDMA) is used increasingly in wireless communications. It is a form of multiplexing communications, e.g., between cellular phones and base stations, based on distinct digital codes in the communication signals. This can be contrasted with other wireless protocols, such as frequency-division multiple access and time-division multiple access, in which multiplexing is based on the use of orthogonal frequency bands and orthogonal time-slots, respectively.

A limiting factor in CDMA communication and, particularly, in so-called direct sequence CDMA (DS-CDMA), is the interference between multiple simultaneous communications, e.g., multiple cellular phone users in the same geographic area using their phones at the same time. This is referred to as multiple access interference (MAI). It has effect of limiting the capacity of cellular phone base stations, since interference may exceed acceptable levels -- driving service quality below acceptable levels -- when there are too many users.

A technique known as multi-user detection (MUD) reduces multiple access interference and, as a consequence, increases base station capacity. MUD can reduce interference not only between multiple signals of like strength, but also that caused by users so close to the base station as to otherwise overpower signals from other users (the so-called near/far problem). MUD generally functions on the principle that signals from multiple simultaneous users can be jointly used to improve detection of the signal from any single user. Many forms of MUD are known; surveys are provided in Moshavi, "Multi-User Detection for DS-CDMA Systems," IEEE Communications Magazine (October, 1996) and Duel-Hallen et al, "Multiuser Detection for CDMA Systems," IEEE Personal Communications (April 1995). Though a promising solution to increasing the capacity of cellular phone base stations, MUD techniques are typically so computationally intensive as to limit practical application.

An object of this invention is to provide improved methods and apparatus for wireless communications. A related object is to provide such methods and apparatus for multi-user detection or interference cancellation in code-division multiple access communications.

A further object of the invention is to provide such methods and apparatus as can be cost-effectively implemented and as require minimal changes in existing wireless communications infrastructure.

A still further object of the invention is to provide methods and apparatus for executing multi-user detection and related algorithms in real-time.

A still further object of the invention is to provide such methods and apparatus as manage faults for high-availability.

Summary of the Invention

These and other objects are met by the invention which provides, in one aspect, a wireless communications system, referred to as the "MCW-1" (among other terms) in the materials that follow, and methods of operation thereof. An overview of that system is provided in the document entitled "Software Architecture of the MCW-1 MUD Board," immediately following this Summary. A more complete understanding of its implementation may be attained by reference to the other attached materials.

In view of those materials, aspects of the invention include, but are not limited to, the following:

- hardware and/or software architectures (and methods of operation thereof) for multi-user detection in wireless communications systems and particularly, for example, in a wireless communications base station;
- a hardware architecture (and methods of operation thereof) for multi-user detection in wireless communications systems pairing each processing node with NVRAM and watchdog PLD for fault management;
- methods and apparatus for connecting watchdog PLDs with an out-of-band fault-management bus;
- methods and apparatus for use of an embedded host with the RACEway™ architecture of Mercury Computer Systems, Inc.
- methods and apparatus for interfacing a digital signal processor to the RACEway™ architecture;

- methods and apparatus for interfacing the RACEway™ architecture to a programming port in a device for multi-user detection in wireless communications systems;
- methods and apparatus for implementing a DMA Engine FPGA for use in multi-user detection in a wireless communications systems;
- methods and apparatus for implementing a hardware-based reset voter and stop voter;
- methods and apparatus for scalable mapping of handset and BTS functions to multiple processors;
- methods and apparatus for facilitating allocation and management of buffers for interconnecting processors that implement the aforementioned mapping;
- methods and apparatus for implementing a hybrid operating system, e.g., with the VxWorks operating system (of WindRiver Systems, Inc.) on a host computer and the MC/OS operating system on RACE®-based nodes. (Race and MC/OS are trademarks of Mercury Computer Systems, Inc.);
- methods and apparatus for high-availability multi-user detection in wireless communications systems, including (by way of non-limiting example) round-robin fault testing and use of NVRAM to store fault symptoms and use of master to diagnose faults from NVRAM contents;
- class library-based methods and apparatus for facilitating interprocessor communications, by way of non-limiting example, in buffering for multi-user detection in wireless communications systems;

- methods and apparatus for implementation of R-matrix, gamma-matrix and MPIC computations on separate processors in a device for multi-user detection in wireless communications systems;

- methods and apparatus for computing complementary R-matrix elements in parallel using multiple processors in a device for multi-user detection in wireless communications systems;

- methods and apparatus for depositing results of R-matrix calculations contiguously in memory in a device for multi-user detection in wireless communications systems;

- methods and apparatus for increasing the number of MPIC and R-matrix calculations performed in cache in a device for multi-user detection in wireless communications systems;

- methods and apparatus for performing a gamma-matrix calculation in FPGA in a device for multi-user detection in wireless communications systems;

- methods and apparatus for equalizing load of R-matrix-element calculation among multiple processors in a device for multi-user detection in wireless communications systems; and

- methods and apparatus for use of Altivec registers and instruction set in performing MUD calculations in a wireless communications system.

These and other aspects of the invention (including utilization of the aforementioned methods and aspects for other than wireless communications and/or interference cancellation) are evident in the materials that follow.

Detailed Description of the Invention

(see attached materials)

Software Architecture of the MCW-1 MUD Board

Table of Contents

Table of Figures

1 Purpose

The purpose of this document is to describe the software architecture of the MCW-1 board. The MCW-1 application is a digital signal processing application that performs interference cancellation for a cellular base station modem board.

The software project consists of 3 major parts:

- Support for the custom MCW-1 board being designed by the Wireless Communications Group hardware department. This consists of porting the existing host (VxWorks) and multicomputer (MC/OS) software to the board, and adding code to support specialized features of the board such as LED control, voltage monitoring, hardware watchdogs, etc.
- Increasing the MTBF of the system by addition of high availability software. This software includes monitoring features such as watchdogs, fault detection/repair algorithms, and remote software download.
- Implementation of the application software. This includes optimal implementation of the MUD algorithms, as well as implementing degraded versions of the algorithm that can be executed when some of the computational hardware is unavailable due to failures.

Detailed information on the design of new software for the MCW-1 board can be found in the appropriate functional design documents, which are listed in the References section of this document.

2 Glossary

1. MTBF – Mean Time Between Failures
2. MUD – Multi User Detection. A class of algorithms to detect multiple interference sources and remove those effects from the signal.
3. Multicomputer – a parallel computer which achieves it's increase in performance by having more than one CPU working on the application simultaneously.
4. VxWorks – a proprietary real time operating system sold by Wind River, Inc.

3 Application Execution Environment

3.1 Overview

The purpose of the MUD application is to input raw antenna data from the base station modem card, detect sources of interference, produce a new stream of data which has had interference removed, and then output the data to the modem card for further processing.

Characteristics of this processing ~~are~~ are that it must have low latency (< 300 microseconds), ~~and~~ must deal with large amounts of data (> 110 million bytes of data per second), and must be very reliable.

The Mercury computer system is well suited to this kind of signal processing, exhibiting both very low latencies and high bandwidths.

The system hardware and software were not designed with high availability as a goal, so reliability is in line with other standard computer systems designed for commercial applications

Input data flows from the Modem Motherboard, over the PCI bus, through the PXB++ bridge, onto the fabric, through the crossbar, and into the memory of the computing elements. Output data flows in the opposite direction. Some data will also flow between the 8240 Host CPU and the compute elements, via a similar pathway, i.e. from the PCI bus through the PXB++ and thus onto the fabric.

Although the software tries to treat the system as if the hardware were symmetric, as can be seen in the following figure, the host 8240 CPU is attached via the PCI bus, not directly to the fabric.

Error! Not a valid link.

Figure 1

3.2 *Operating System*

MC/OS was selected as the operating system for the MCW-1 board because it provides the low latencies and high I/O and IPC bandwidths required for these sorts of algorithms, and also because it already provides support for most of the hardware being incorporated on the MCW-1 board.

The MUD application can be kept as portable as possible by minimizing the use of non-POSIX MC/OS system calls, and encapsulating calls into proprietary MC/OS interfaces such as DX.

MC/OS requires the presence of a host computer system, which in this case will be a Motorola 8240 PowerPC processor running the VxWorks operating system.

Software Architecture of the MCW-1 MUD Board

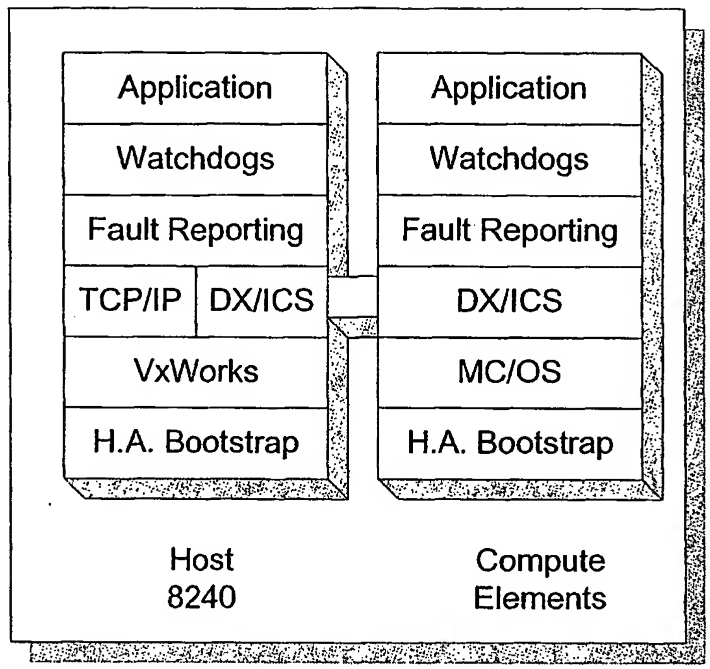


Figure 2

3.3 IPC

The MC/OS DX subsystem will be used for IPC within the application. This API provides low overhead, low latency access to the Mercury DMA engines, which in turn provide high bandwidth transfers of data. DX will be used to move data between the G4 compute elements during parallel processing, and also will be used to move data between the MC/OS compute elements, the VxWorks host computer, and the motherboard modem card.

3.4 I/O

Input / Output between the MUD card and the motherboard modem card takes place by moving data between the Race++ Fabric and the PCI bus via the PXB++ bridge. The application will use DX to initialize the PXB++ bridge, and to cause input/output data to move as if it were regular DX IPC traffic.

Discussions with the customer need to take place in order to determine exactly how data flows over the PCI bus. For instance, it is currently unclear who will initiate data transfers, and how the initiator will know which PCI addresses should be involved in the transfer. A number of meetings with the customer are required to resolve these issues.

3.5 High Availability

The approach to high availability on the MCW-1 card is to do most of the high availability processing at a time when the application is not running. Specifically, faults are handled by rebooting the system (fairly quickly). When the system comes up, the application can determine which processing resources are available, and it is up to the application to determine how to map its processing needs onto the available resources.

This approach to high availability means that there are short interruptions in service, but that the application does not need to know how to continue execution across faults. For instance, the application can make the assumption that the hardware configuration will not change without the system first rebooting.

If the application has state which needs to be preserved across reboots, the application is responsible for checkpointing the data on a regular basis. The system software will provide an API to a portion of the non-volatile RAM for this purpose. It should be noted that the non-volatile RAM is quite small, and that storage of more than a few hundred bytes of data will require another mechanism to be put in place.

4 Operating System Environment

4.1 Overview

Mercury Computer Systems, Inc. has historically had the concept of a host computer system. This dates back to the days when Mercury produced array processors that were attached to customers' mainframe computers. The evolution of Mercury multicomputers has left a vestigial host that often performs little more service than as a bootstrap device for the multicomputer.

The host computer system survives in the MCW-1 design primarily as a way to reduce schedule risk. The existence of a host computer system is assumed in so many ways by the existing Mercury software, that it would add significant schedule risk to attempt to remove this assumption in the MCW-1 timeframe.

In the MCW-1 board, the host system performs the following functions:

- It configures the Compute Elements, Fabric, and Bridges
- It loads executable code into the Compute Elements
- It serves as a bridge to the TCP/IP internetwork
- It serves as a file system daemon
- It runs some of the application software
- It manages some of the specialized high availability hardware

4.2 Bootstrap

The host computer system is based on a Motorola 8240 PowerPC processor on the MCW-1 board. The 8240 is attached to an amount of linear flash memory. This flash memory serves several purposes.

The first purpose the flash memory serves is as a source of instructions to execute when the 8240 comes out of reset. **Linear** flash is flash which can be addressed as if it was normal RAM. Flash memories can also be organized to look like disk controllers; however in that configuration they require a disk driver to

provide access to the flash memory. Although such an organization has several benefits such as automatic reallocation of bad flash cells, and write wear leveling, it is not appropriate for initial bootstrap.

The flash memory also serves as a file system for the host (see Section 4.6), and as a place to store board permanent information (such as a serial number). Refer to the function design specification (TBS) for more details on how flash memory is used.

When the 8240 first comes out of reset, memory is not turned on. Since high level languages such as C assume some memory is present (for a stack, for instance), the initial bootstrap code must be coded in assembler. This assembler bootstrap should only be a few hundred lines of code, sufficient to configure the memory controller, initialize memory, and initialize the configuration of the 8240 internal registers.

After the assembler bootstrap has finished execution, control is passed to the MCW-1 H.A. code (which is also contained in boot flash memory). The purpose of the H.A. code is to attempt to configure the fabric, and load the compute element CPUs with H.A. code. Once this is complete, all the processors participate in the H.A. algorithm. The output of the algorithm is a configuration table which details which hardware is operational and which hardware is not. This is an input to the next stage of bootstrap, the **Multicomputer Configuration.**

4.3 Multicomputer Configuration

MC/OS expects the host computer system to configure the multicomputer. The configmc program reads a textual description of the computer system configuration, and produces a series of binary data structures that describe the computer system configuration. These data structures are used in MC/OS to describe the routing and configuration of the multicomputer.

The MCW-1 board will use almost exactly the same sequence to configure the multicomputer. The major difference is that MC/OS expects configurations to be totally static, whereas the MCW-1 configuration will need to change dynamically as faulty hardware cause various resources to be unavailable for use.

There are currently two proposals being considered for how this dynamic reconfiguration takes place.

The first proposal is that the binary data structures produced by configmc are modified to include flags that indicate whether a piece of hardware is usable or not. A modification to MC/OS would prevent it from using hardware marked as broken. The risk here is that the modifications to MC/OS may be non-trivial. The benefit may be faster reboot times.

The second proposal is that the output of the H.A. algorithm is used to produce a new configuration file input to configmc, the configmc execution is repeated with the new file, and MC/OS is configured and loaded with no knowledge of the broken hardware whatsoever. This proposal has the added benefit that configmc may be able to calculate the most optimal routing tables in the face of failed hardware, minimizing the performance impact of the failure on the remaining components. This proposal provides risk reduction given that MC/OS changes would not be required.

4.4 *Multicomputer Loading*

After the host computer has configured the multicomputer, the **runmc** program loads the functional compute elements with a copy of MC/OS. The only changes required for the MCW-1 board is for the loading process to examine which hardware may be offline because it is faulty, and take this into account when determining which compute elements need to be loaded.

4.5 *TCP/IP Bridge*

We believe that the customer is likely to require access to the MCW-1 board from a TCP/IP network. MC/OS nodes do not contain a TCP/IP stack; therefore the host computer system acts as a connection to the TCP/IP network. The VxWorks operating system contains a fully functional TCP/IP stack. All currently envisioned daemons that need access to the TCP/IP network will run on the host processor. Should the need arise for compute elements to access network resources, the host computer would have to act as a proxy, exchanging information with the compute element utilizing DX transfers, and then making the appropriate TCP/IP calls on behalf of the compute element.

4.6 *File System*

The host computer system needs a file system to store configuration files, executable programs, and MC/OS images. Rotating disks have insufficient MTBF times; therefore flash memory will be utilized. Rather than have a separate flash memory from the host computer boot flash, the same flash is utilized for both bootstrap purposes and for holding file system data. A commercial flash file system will be purchased and ported which provides DOS file system semantics as well as write wear leveling. Wear leveling attempts to spread the number of writes evenly across the sectors of flash memory, as flash memory can only be written a finite number of times before it is worn out. Modern flash devices can be written around 100,000 times before they are worn out.

4.7 *Remote Software Upgrade*

The current design of the MCW-1 board assumes that the customer will want to update system and application code in the field, via network. There are two portions of code which need to be updated – the bootstrap code which is executed by the 8240 processor when it comes out of reset, and the rest of the code which resides on the flash file system as files.

When code is initially downloaded to the MCW-1, it is written as a group of files within a directory in the flash file system. A single top level file keeps track of which directory tree is used to boot the system. This file continues to point at the existing directory tree until a download of new software is successfully completed. When a download has been completed and verified, the top-level file is updated to point to the new directory tree, the boot flash is rewritten, and the system can be rebooted.

A possible problem in multi-board systems is how to deal with different versions of released software on different boards. For instance, if board 1 has revision 1.0 of the software distribution, and board 2 has revision 1.1 of the software distribution, will the two versions work together, or will there be a way

to ensure that the same version of software is installed on all boards. This issue does not occur on the MCW-1 because it is a single board solution; therefore this issue can be addressed at a later time.

A commercial solution to remote software upgrade is available, and has been ported to VxWorks. It is our intent to port this code at a future date.

5 High Availability

5.1 Goals

The goal of the high availability features of the MCW-1 is to increase the MTBF of the system as much as possible with little or no increase in cost to the board. The requirement for minimal cost increase rules out such common approaches as hot or cold standby, replicated hardware, etc.

It is not a goal to provide uninterrupted computing during hardware or software failures, nor is it a goal to provide fault tolerance.

~~1.2~~5.2 Fault Detection & Isolation

Fault detection is performed by having each CPU in the system gather as much information about what it observed during a fault, and then comparing the information in order to detect which components could be the common cause of the symptoms. In some cases, it may take multiple faults before the algorithm can detect which component is at fault. The requirement not to add expensive hardware for fault detection means that in many cases the algorithm will not be able to determine which component is at fault.

The MCW-1 board has many single points of failure. Specifically, everything on the board is a single point of failure except for the compute elements. This means that the only hard failures that can be configured out are failures in the compute elements. However, many failures are transient or soft, and these can be recovered from with a reboot cycle. Therefore, we expect the high availability features to have a positive effect on the MTBF of the card.

More detailed information is available in the functional design specification (1).

5.3 Degraded Application

In the case of hard failures of a compute element, the application will have to execute with reduced demand for computing resources. There are several strategies possible for the MUD algorithm to decrease computing demands, such as working with a smaller number of interference sources, or performing a less complete job of interference cancellation.

We expect the computing requirements of the algorithm to be high enough that failure of more than a single compute element will cause the board to be inoperative. Therefore, the MCW-1 application only needs to handle two configurations: all compute elements functional and 1 compute element unavailable. We believe that a small amount of startup code can map the application onto the two possible configurations. Note that the single crossbar means that there are no issues as to which processes need to go on which processors – the bandwidth and latencies for any node to any other node are

identical on the MCW-1. This will not be true of larger systems in the future, and we will eventually need a way to map computing and I/O requirements onto arbitrary hardware configurations.

5.4 Remote Software Upgrade

Downtime due to the updating of software is counted against the availability of a computer system, and therefore a remote reload of software is a necessity. The MCW-1 is capable of downloading new software during normal operation. The reboot strategy means that the downtime due to starting up new software is only a few seconds.

Referenced Documents

1. "MC/OS High Availability Functional Design Specification", Yevgeniy Tarashchanskiy, 17 April, 2000.

Mercury Computer Systems

Wireless Communications

Hardware Engineering

MCW-1a Functional Specification

Memorandum #SRI-1

31 January 2001

Revision 3.00

This document was created using MS Word 97 and is located at
TBD

Notice: If you are not viewing this document in electronic form at the above full path-name, it is not guaranteed to be the latest revision.

MCW-1a Functional Specification

1 REVISION HISTORY

Revision 0.0 - 3/17/00 Steven Imperiali
Initial Entry

Revision 0.01 - 4/25/00 Steven Imperiali
Minor corrections, filled in missing sections.

Revision 0.1 - 5/5/00 Steven Imperiali
Incorporated review comments.

Revision 0.2 - 5/8/00 Steven Imperiali
Incorporated review comments.
Removed reference to RapidIO/Race++ bridge
Revision 0.21 - 5/16/00 Steven Imperiali
Incorporated review comments.
Modified MPC8240 memory map
Revision 0.22 - 5/26/00 Steven Imperiali
Modified MPC8240 Memory Map

Revision 1.00 - 7/24/00 Steven Imperiali
Modified MPC8240 Memory Map
Updated memo with current design status

Revision 2.01 - 11/01/00 Steven Imperiali
Modified power supply ramp requirements

Revision 2.02 - 11/15/00 Steven Imperiali
Modified interrupt controller

Revision 2.03 - 1/26/01 Steven Imperiali
Minor documentation corrections

Revision 3.00 - 1/31/01 Steven Imperiali
Modified memo to reflect MCW-1a modules

2 REFERENCE DOCUMENTS

1. American National Standard for RACEway Interlink (ANSI/VITA 5-1994)
2. PCI Rev 2.2 Local Bus Specification
3. PCE133 ASIC Hardware Specification
4. XBAR++ Function Specification
5. PXB++ PCI Bridge Functional Specification
6. PowerPC 7400 PPC Microprocessor Hardware Specification
7. Flash Memory Specification p/n TBD
8. MCW-1Product Definition Document (PDD) vTBD
9. Technical brief of Mercury Computer Systems RACE++ series topologies
10. MPC8240 Users Manual (MPC8240UM/D 07/1999 Rev. 0)

3 MERCURY PART NUMBER

The board identifier name is MCW-1a and the Mercury part number is 560549.

4 FUNCTIONAL DESCRIPTION

4.1 OVERVIEW

The MCW-1a is designed to be an algorithm processing daughter card utilizing the MPC7400 PPC, MPC8240, PCE133 ASIC, XBAR++ ASIC, and PXB++ FPGA. The MCW-1mates with a Motorola base station modem board. MCW-1a can provide additional connectivity between processing elements in different sector slots utilizing over-the-top RACEway++ cables. It is a Motorola form factor card with four computational nodes and one host node. The computational nodes (CNs) are based on the latest MPC7400 PPC microprocessor and the host is an MPC8240. The MCW-1can provide one Ethernet 10/100 BT port on the front panel. A 32-bit, 66 MHz PCI interface provide the interface to the Motorola board.

The MCW-1a block diagram is shown in Figure 1.

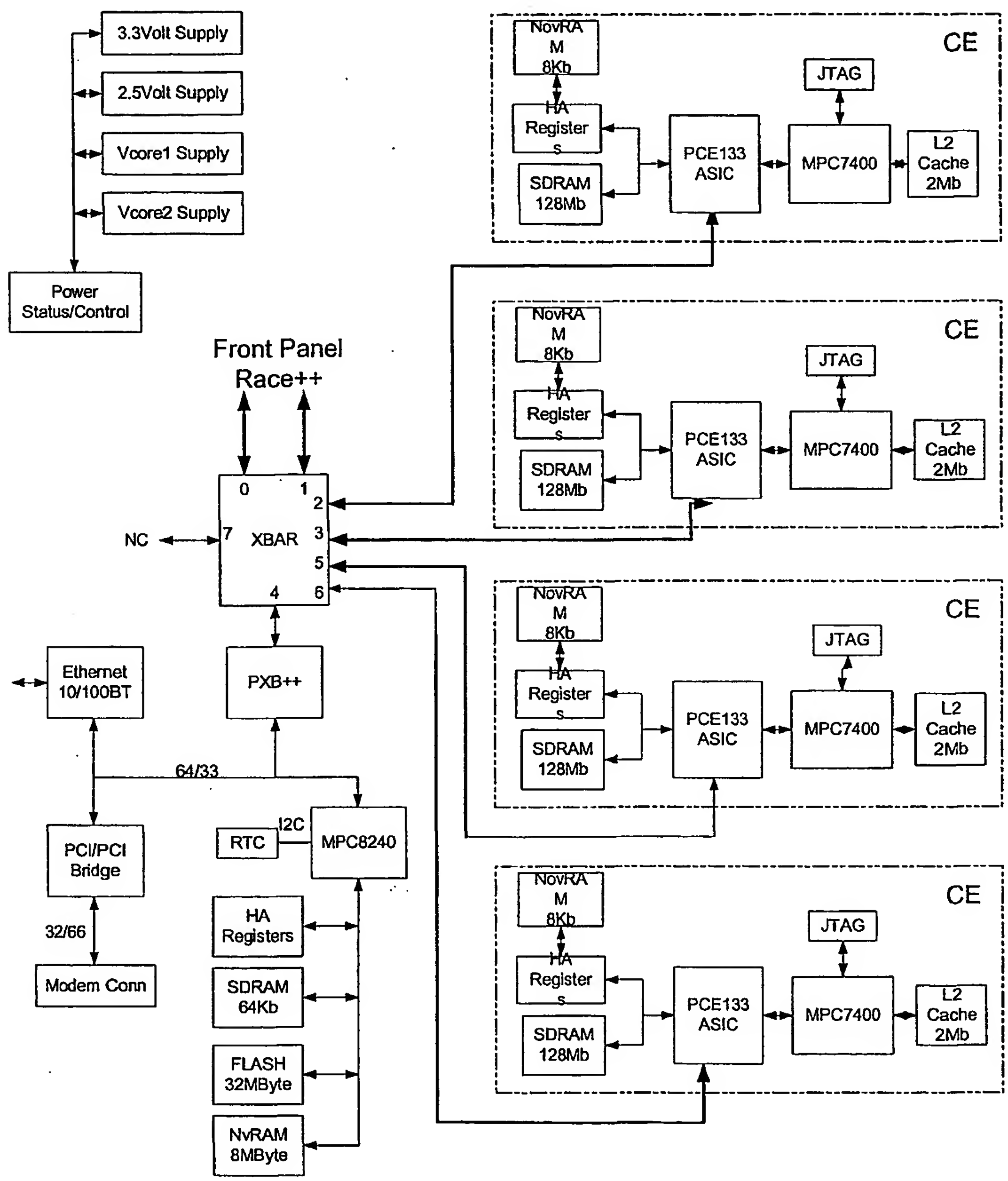


Figure 1. MCW-1A BLOCK DIAGRAM

MCW-1a Functional Specification

Figure 2 shows the MCW-1a system topology. Table 1 gives the proposed route codes for the board.

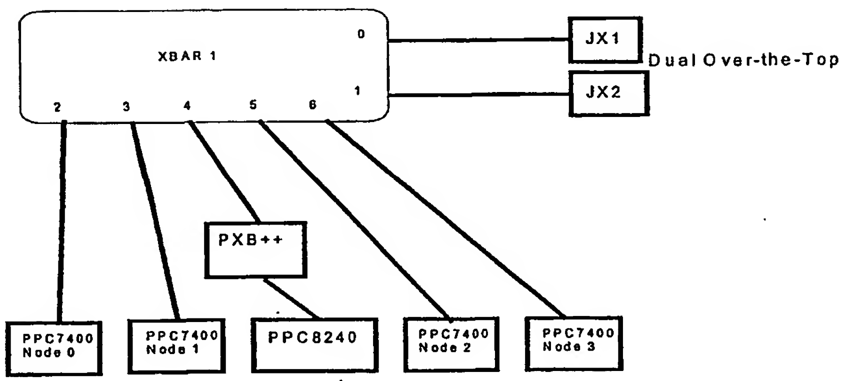


Figure 2. MCW-1A BOARD-LEVEL TOPOLOGY

Table 1. Route Codes for MCW-1a Board XBAR

Route Code	Destination for Virtual Ports	Physical XBAR 1 Ports
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

4.2 FEATURES

- Custom size daughter card
- Master PCI 32-bit @ 66MHz compliant with REV2.2 PCI local bus spec.
  PCI write peak performance is 240 MB/sec.
  PCI read peak performance is 220 MB/sec.
- Single IEEE802.3 compliant Ethernet 10BASE-T//100BASE-T
- Four computation nodes (CNs) based on MPC7400 PPC running @ 400 MHz.
  1 MB L2 cache per CN @ 200 MHz to 266 MHz.
  128 MB SDRAM with ECC per CN @ 133 MHz.
  Hardware based watchdog monitor.
  One PCE133 ASIC per CN.
- Two, over-the-top, 66 MHz RACEway++ interlink ports
  configured in cable mode.
- PCI interface 32-bit @ 66 MHz.
- RACEway++ crossbar to connect nodes.
- PXB++ 64-bit @ 33 MHz PCI bus.
- Non-transparent 64-bit/33 MHz to 32-bit/66 MHz PCI bridge.
- 200MHz PPC8240 PowerPC processor.
  32-bit 33MHz PCI bus.
  100MHz, 64Mbytes SDRAM.
- Bulk FLASH interface.
  Linear address mode.
  32 banks of 1Mbytes.
- LEDs.
- 8Kbytes non-volatile SRAM.
- Real time clock.
- Compute node fault isolation control.
- JTAG test port.

4.3 CONFIGURATION OPTIONS

4.3.1 CPU Options

- MPC7400 @ 400 MHz.
- MPC7410 @ 400 MHz.

4.3.2 SDRAM Options

- 128 MB SDRAM @ 133 MHz with ECC.

4.3.3 FLASH Memory Options

- 16 MB FLASH memory.
- 32MB FLASH memory

4.3.4 Ethernet Options

- No Ethernet.
- Simgle Ethernet

4.4 REQUIREMENTS

4.4.1 Mechanical Form Factor

The MCW-1a form factor conforms to TBD Motorola mechanical requirements.

4.4.2 Power Requirements

The MCW-1a requires +5.0 volts from the modem board. The +1.5V to +2.1V MPC7400 core voltage required by the core of MPC7400 is converted from +5.0V on the board. There are two core supplies used to power the four cpu cores. The 2.5V voltage required is converted from +5.0V by an onboard power supply. The 3.3V voltage required is also converted from +5.0V by an onboard power supply. The MCW-1a estimated typical power dissipation is 50 watts @ 5.0V.

4.4.3 Electrical Interface

The MCW-1a provides a PCI 32-bit, 66 MHz interface to the Motorola modem board via an 80-pin connector.

The MCW-1a provides two over-the-top RACEway++ ports via two connectors located on the front panel.

The MCW-1a provides the single Ethernet 10/100 BT interface available from one RJ-45 connector. The Ethernet interface is provided by a third party Ethernet-to-PCI interface controller chip that is bridged to the crossbar RACEway++ port by means of a PXB++ FPGA (See Figure 2).

4.4.4 Functional

1. Shall have the Main SDRAM memory at 133MHz or greater.
2. Shall have a 1Mbyte L2 Cache at 200MHz or greater.
3. All CE nodes shall have 128Mbyte of SDRAM.
4. Host node shall have at least 32Mbytes of nonvolatile memory.

Form factor requirements:

5. Shall be a daughter card that is ¾ of a Motorola proprietary form factor modem payload card sized 11" by 14". On 20mm centers board to board.{actual shape, dimensions etc TBD via drawings from Motorola.}
6. Shall be electrically a PMC module, TBD from further discussions with customer.
7. Shall use P1, P2 for 32/66MHz PCI bus.
8. Shall have a maximum heat dissipation of 50W

System requirements

9. A minimum of 105Mbyte/sec from the modem payload module to the MCW-1a card shall be provided through the PCI interface.
10. From the MCW-1a card to Motorola Modem Payload module output bandwidth shall be at least 200kbyte/sec, concurrent with the 105Mbyte/sec input.
11. The system shall have a bandwidth of at least 250Mbyte/sec between CE's, e.g. RACE++ at 66Mhz, as a minimum.
12. Shall have non-volatile memory, for at least 32Mbytes of data.
13. Shall support software upgrade from remote locations.

4.5 COMPATIBILITY

The MCW-1a board is a custom daughter card designed for the Motorola base station modem board.

4.6 PERFORMANCE

The PCI bus standard and the PXB++ FPGA limits the RACEway++ to the PCI performance. Peak transfers of 240 MB/sec are achievable between the PXB++, PPC8240 and the non-transparent PCI Bridge. (See Figure 1)

Data transfers of up to 266 MB/sec peak are supported for access from RACEway++ to/from the MPC7400 CE's local SDRAM memory.

PCE133 ASIC-initiated DMA transfers run at optimum RACEway++ speeds approaching 266 MB/sec peak. Data can be transferred with the DMA from a single DMA command transfer to/from the CN's local SDRAM memory to/from RACEway++. The DMA engine formats transfers across RACEway++ optimally using packets up to 2048 Bytes.

The operating clock frequency of the PCE133 ASIC, SDRAM, and MPC7400 processor bus is 133 MHz. Likewise, the operating frequency for the RACEway++ is 66 MHz. The local PCI clock is used by the corresponding PXB++ FPGA and does not exceed 33 MHz.

A separate 25 MHz oscillator is included on the MCW-1a for driving the Ethernet interface.

4.7 DETAILED DESCRIPTION

The MCW-1a block diagram is shown in Figure 1.

4.7.1 Modem Board Interface

TBD (PCI 32-bit 66MHz).
TBD PCI to PCI bridge stuff.
TBD Motorola requirements.

4.7.2 Board Resets

There are several sources of reset to the daughter card. A MAX823 voltage supervisor will generate a 200ms reset after VCC rises above 4.38 volts. When the MAX823 reset is deasserted, state machine logic will monitor PCI_RESET_0. The state machine will continue driving RESET_0 until both the MAX823 and PCI_RESET_0 are deasserted. Either reset will generate the signal RESET_0 which will reset the card into its power-on state. RESET_0 will also generate the HRESET_0 and TRST signals to the five CPUs. HRESET_0 and TRST for each of the cpus can also be generated by their JTAG ports; JTAG_HRESET_0 and JTAG_TRST respectively. The MCP8240 is capable of generating a reset request, a soft reset (C_SRESET_0) to each CPU, a checkstop request, and a CE ASIC reset (CE_RESET_0) to each of the four CE ASICs. A discrete from the 5v powered reset PLD will generate the signal NPORESET_1 (not a power on reset). This signal is fed into the MPC8240's discrete input word. The MPC8240 will read this signal as a logic low only if it is coming out of reset due to either a power condition or an external reset from offboard. Each node, as well as the MPC8240 may request a board level reset. These requests are majority voted, and the result RESETVOTE_0 will generate a board level reset

Figure 3 shows the MCW-1a hard reset generation function

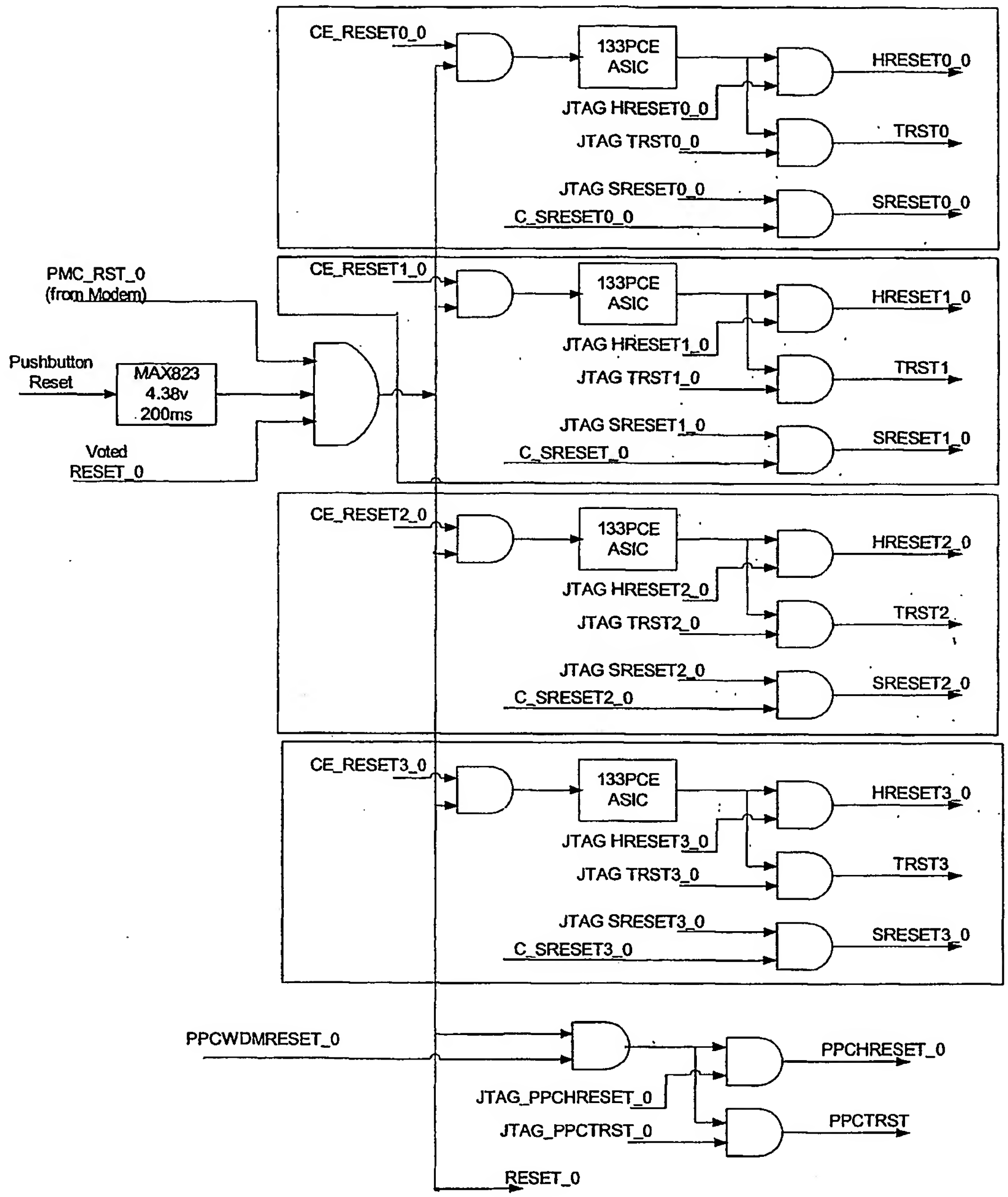


Figure 3. **HARD RESET FUNCTIONAL BLOCK DIAGRAM**

4.7.3 Watchdog Monitor

There are five independent watchdog monitors on the MCW-1a card. Each processor node is responsible for strobing its watchdog once every 20 msec (initial window after board level reset is 2 sec) but no sooner than 500 usec. Strobing the watchdog for the processing nodes is accomplished by writing a zero/one sequence to the DIAG3 discrete coming from the PC133PCE ASIC. The MPC8240's watchdog is serviced by writing to the memory mapped discrete location FFFF_D027. A single write of any value will strobe the watchdog. Upon power-on, the watchdogs come up in a failed state; once a valid strobe is issued; the watchdog will be satisfied. If the CPU fails to service the watchdog within the valid window, the watchdog will fail. A watchdog of a failing processing node will trigger an interrupt to the MPC8240. An MPC8240 watchdog fault will trigger a reset to the board. The watchdog will then remain in a latched failed state until a CPU reset occurs followed by a valid service sequence. Figure 4 shows a valid service sequences of the watchdog.

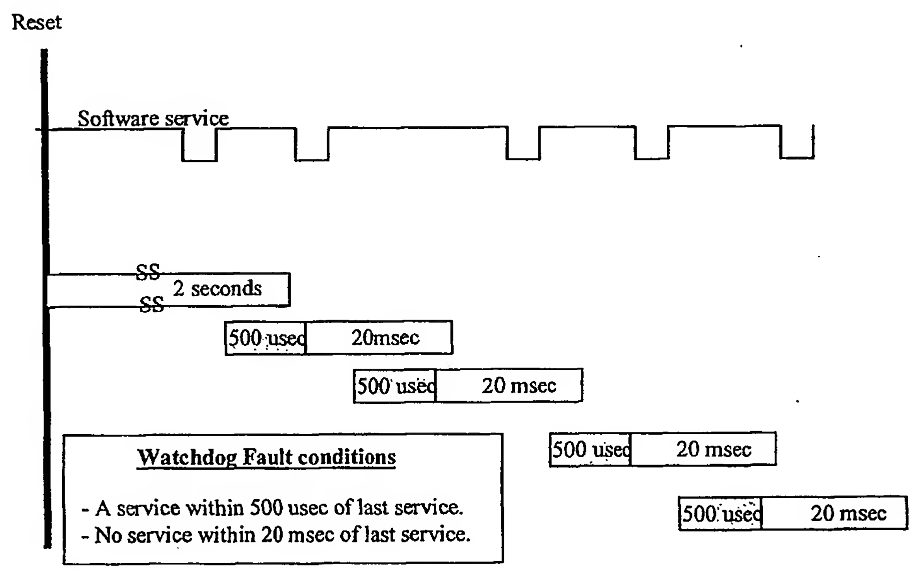


Figure 4. EXAMPLE WATCHDOG SERVICE SEQUENCES

4.7.4 Operating Frequency

The MPC7400 bus runs at 133 MHz. The L2 cache bus of the MPC7400 runs at 200 MHz to 266 MHz. The SDRAMs run at 133 MHz. The RACEway++ interface runs at 66 MHz. The local PCI bus runs at 33 MHz and the off board PCI runs at 66MHz. The MPC8240's internal frequency is 200 MHz while its SDRAM interface is 100 MHz.

4.7.4.1 Clock Margining

This card has two crystal oscillators for the three clock domains present on the card, a 66 MHz oscillator for the RACEway++ interface and MPC7400 CNs. The 66MHz frequency is divided in half to generate a 33 MHz signal for the PCI interface. A second oscillator, 25 MHz, clocks the Ethernet and watchdog circuitry. Both the PCI and MPC clocks are marginable. In order to provide clock margining, a 4-pin connector allows the test engineer to functionally disable the onboard oscillator and replace it with a test frequency. The pinout of this connector is detailed in Table 2.

Table 2. Test Clock Connector

Pin	Signal
1	GND
2	/Test Clock
3	Test Clock
4	Test Clock Enable L

4.7.5 Serial Configuration EEPROM

There are several serial EEPROMs used to loadconfiguration to the CE ASICs, PXB++ and XBAR++ after reset. The serial PROM functionality can be found in the ASIC's functional specification.

4.7.5.1 CE ASIC Serial EEPROM

The serial EEPROM can be read and programmed by means of the RACEway++ bus. It is programmed during manufacture of the MCW-1a to contain configuration information for CE ASIC. The serial EEPROM AT24C128 is controlled from the CE ASIC. After reset, the CE ASIC automatically reads the first location from the serial EPROM. Refer to the CE ASIC functional specification, reference 3, for information on reading and writing this device.

4.7.5.2 PXB++ FPGA Serial EEPROM

The serial EEPROM can be read and programmed by means of the PCI bus or the RACEway++ bus. It is programmed during manufacture of the MCW-1a to contain configuration information for PXB. The serial EEPROM AT24C128 device is 128K bits and is controlled from the PXB++. After reset, the PXB++ automatically reads 8 KB from the serial EEPROM and initializes the PXB++ internal registers. Refer to the PXB++ FPGA functional specification, reference 5, for information on reading and writing this device.

4.7.5.3 XBAR++ ASIC Serial EEPROM

The serial EEPROM can be read and programmed by means of the RACEway++ bus. It is programmed during manufacture of the MCW-1a to contain configuration information for XBAR++. The serial EEPROM AT24C128 is controlled from the XBAR++ ASIC. After reset, the XBAR++ ASIC automatically reads from the serial EPROM and initializes the XBAR++ internal registers. Refer to the XBAR++ ASIC functional specification, reference 4, for information on reading and writing this device.

4.7.5.3.1 Register Description

Reference 4 f describes the registers of the XBAR++ ASIC.

4.7.6 RACEway++ Interconnect

Communication between all processing and I/O elements on the system card is provided by a Mercury eight-port crossbar XBAR++ ASIC. The XBAR++ provide up to three simultaneous 266 MB/sec peak throughput data paths between elements for a total peak throughput of 798 MB/sec. Three crossbar ports connect to the RapidIO Bridge FPGA. Each MPC7400 CN uses one crossbar port. The Ethernet and MPC8240 interface to a crossbar port through the PXB++. (See 0) Reference 4 describes the operation and registers of the XBAR++ ASIC.

4.7.7 Local PCI I/O Bus

The PXB++ FPGA provides the local PCI I/O bus. This bus is accessible by means of the RACEway++ from the processing nodes. All resources on this bus are initialized and controlled by the MPC8240. This bus provides access to an Ethernet controller, PCI to PCI transparent bridge and the PPC8240 host controller. Transfers from devices on this local PCI bus to and from devices on the RACEway++ can achieve 240 MB/sec for writes and 220 MB/sec for reads. These rates assume block transfers of reasonable size.

4.7.7.1 PXB++ Program EEPROM

The PXB++ FPGA is programmed by an XC18V04 configuration EEPROM running in parallel mode. Configuration initiates when a power-on or board level reset occurs. Dividing the onboard 33MHz generates the configuration clock of 16.6MHz. The configuration EEPROM itself is onboard programmable through the JTAG scan chain.

4.7.7.1.1 Register Description

Reference 5 describes the registers of the PXB++ ASIC.

4.7.8 Ethernet Interface

The PCI-to-Ethernet interface uses the AM79C973 Pcnet-FAST III single chip 10/100 Mbps Ethernet controller. This device is equipped with a built in physical layer interface to achieve a minimal parts count Ethernet interface. A 25 MHz oscillator provides the proper clock frequency to the Ethernet chip. The PCI interrupt from the Ethernet chip is wired to the MPC8240's external interrupt controller.

4.7.9 MPC7400 or Nitro Computer Nodes (CNs)

The board contains four MPC7400 CNs. Each MPC CN uses a PCE133 ASIC to interface the cpu to RACEway++. The PCE133 ASIC provides all the standard features of a CN, such as a DMA engine, mail box interrupts, timers, RACEway++ page mapping registers, SDRAM interface, and so on. Local memory for each CN consists of 32, 64, or 128 MB SDRAM, and L2 cache SRAM. Each CN also has a nonvolatile SRAM and watchdog monitor. The cpu bus is 64-bit data, 32-bit address, and operates synchronously at 133 MHz.

4.7.9.1 Processor

The MCW-1a card is designed to use either the 400 MHz MPC7400 or the 400 MHz Nitro processors. The processor is packaged in a 25mm, 360-ball CBGA package. Each processor requires the attachment of a heat sink to keep it within its thermal limits.

4.7.9.2 MPC7400 L2 Cache

The MPC7400 L2 cache for each CN is composed of pipelined, single-cycle deselect, sync burst SRAM. This is implemented using two 64K, 128K, or 256K by 36-bit sync burst SRAM parts to make a 0.5 MB, 1 MB, or 2 MB L2 cache. MPC7400 L2 cache can be depopulated to 0 MB.

4.7.9.3 PCE133 ASIC

The MPC processor compute element ASIC (PCE133 ASIC) is a Mercury-designed component. It provides the interface between the MPC7400, the synchronous DRAM, and the RACEway++. All the PCE133 features such as DMA, mailbox interrupts, timers, address snooping, prefetch buffers, and so on, are available in this configuration. This chip is provided in a 35mm, 388-ball BGA package. Reference 3 describes the operation and registers of the PCE133 ASIC.

4.7.9.3.1 Register Description

Reference 3 describes the registers of the PCE133 ASIC.

4.7.9.4 Address Map

4.7.9.4.1 Master Address Map

MCW-1a Functional Specification

Transfers from the MPC7400 to the PCE133 ASIC and RACEway++ are address mapped as shown in Table 3. The SDRAM is 8-, 16-, 32-, or 64-bit addressable. RACEway++ locked read/write and locked read transactions are supported for all data sizes. The 16 Mbyte boot FLASH area is further divided in Table 4

Table 3. Master Address Map

From Address	To Address	Function
0x0000 0000	0x0FFF FFFF	Local SDRAM 256 MB
0x1000 0000	0x1FFF FFFF	XBAR 256 MB map window 1
0x2000 0000	0x2FFF FFFF	XBAR 256 MB map window 2
0x3000 0000	0x3FFF FFFF	XBAR 256 MB map window 3
0x4000 0000	0x4FFF FFFF	XBAR 256 MB map window 4
0x5000 0000	0x5FFF FFFF	XBAR 256 MB map window 5
0x6000 0000	0x6FFF FFFF	XBAR 256 MB map window 6
0x7000 0000	0x7FFF FFFF	XBAR 256 MB map window 7
0x8000 0000	0x8FFF FFFF	XBAR 256 MB map window 8
0x9000 0000	0x9FFF FFFF	XBAR 256 MB map window 9
0xA000 0000	0xAFFF FFFF	XBAR 256 MB map window A
0xB000 0000	0xBFFF FFFF	XBAR 256 MB map window B
0xC000 0000	0xCFFF FFFF	XBAR 256 MB map window C
0xD000 0000	0xDFFF FFFF	XBAR 256 MB map window D
0xE000 0000	0xEFFF FFFF	XBAR 256 MB map window E
0xF000 0000	0xFBFF FBFF	Not used (CE reg replicated mapping)
0xFBFF FC00	0xFBFF FDFF	Internal CN ASIC registers
0xFBFF FE00	0xFEFF FFFF	Prefetch control
0xFF00 0000	0xFFFF FFFF	16 MB boot FLASH memory area

Table 4. Boot FLASH Address Map

From Address	To Address	Function
0xFF00 2006	0xFF00 2006	Software Fail Register
0xFF00 2005	0xFF00 2005	MPC8240 HA Register
0xFF00 2004	0xFF00 2004	Node 3 HA Register
0xFF00 2003	0xFF00 2003	Node 2 HA Register
0xFF00 2002	0xFF00 2002	Node 1 HA Register
0xFF00 2001	0xFF00 2001	Node 0 HA Register
0xFF00 2000	0xFF00 2000	Local HA Register (status/control)
0xFF00 0000	0xFF00 1FFF	NovRAM

4.7.9.4.2 Slave Address Map

Slave accesses are defined as accesses initiated by an external RACEway++ device directed toward the MPC7400 CN. The MPC is not accessible as a slave device. The SDRAM is 8-, 16-, 32-, or 64-bit addressable. RACEway++ locked read/write and locked read are supported for all data sizes. The PCE RACEway port supports a 256 MB address space partitioned as follows in Table 5:

Table 5. Slave Address Map

From Address	To Address	Function
0x0000 0000	0x0FFF FBFF	256 MB less 1 KB hole SDRAM
0Xfff_FC00	0xFFF_FFFF	PCE133 internal registers

4.7.9.5 Interrupt

Reference 3 describes the internal interrupt sources for the PCE133 ASIC. The external interrupt pin on the PCE133 ASIC is driven by the HA PLD and is currently not used. The interrupt output from the PCE133 ASIC is wired to the CPU's external interrupt input pin.

4.7.9.6 PCE133 DIAG Bits

The DIAG3 signal is wired to the HA PLD and is used to strobe the nodes hardware watchdog monitor. The DIAG2 signal is wired to the MPC8240's interrupt controller and is used, by the node, to generate a general purpose interrupt to the MPC8240. The DIAGBIT signal is wired to the HA PLD and is currently not used.

4.7.9.7 MPC7400 Reset

The MPC7400 hard reset signal is driven by three sources gated together: the HRESET_0 pin on the PCE133 ASIC, HRESET_0 from the JTAG connector, and HRESET_0 from the majority voter. The HRESET_0 pin from the CE ASIC is set by the "node run" bit field (bit 0) of the PCE133 ASIC's Miscon_A register. Setting HRESET_0 low causes the MPC7400 to be held in reset. HRESET_0 is low immediately after system reset or power-up, the MPC7400 is held in reset until the HRESET_0 line is pulled high by setting the node run bit to 1. The JTAG HRESET_0 is controlled by debugger software when a JTAG debugger module is connected to the card. The HRESET_0 from the majority voter is generated by a majority vote from all healthy nodes to reset.

4.7.9.8 Boot Procedures

When a cpu reset is asserted, the MPC7400 is put into reset state. The MPC7400 will remain in a reset state until the RUN bit 0 of the Miscon_A register is set to 1 and the MPC8240 has released the reset signals in the discrete output word. The RUN bit should be set to 1 after the boot code has been loaded into the SDRAM starting at location 0x0000_0100. The ASIC maps the reset vector 0xFFF0_0100 generated by the MPC7400 to address 0x0000_0100.

4.7.9.9 MPC7400 CN SDRAM

The main memory for each CN is composed of one bank of synchronous DRAM. This is implemented using five K4S280832A-TC/L75 @133 MHz synchronous DRAM parts. As shown in the memory map (See Table 3), the main memory begins at address 0x0 and grows upward in the address space as memory is increased. The PCE133 ASIC supports error correction (ECC) on the SDRAM.

The SDRAM operates as zero wait state memory and can provide up to 1 GB/sec peak bandwidth on writes from MPC7400 and 800 MB/sec peak bandwidth on read from the MPC7400. ECC error correction is supported.

4.7.9.10 MPC7400 Non-Volatile RAM

Each node will be equipped 8Kx8 of non-volatile RAM for the storage of fault record data and configuration information. This function is implemented using a SIMTEK STK12C68S45 NOVRAM attached to the PCE133 ASIC's boot FLASH interface. The data bus of the device is isolated from the PCE ASIC through an IDT IDTQS32244SO buffer. This buffer provides loading isolation and 3.3v to 5v translation.

4.7.10 MPC8240 Host Controller

The MPC8240 integrated processor is comprised of a peripheral logic block and a 32-bit embedded MPC603e PowerPC processor core. The peripheral logic integrates a PCI bridge, memory controller, DMA controller, EPIC interrupt controller, a message unit, and an I2C controller. The processor core is a full featured, high-performance processor with floating-point support, memory management, 16Kbytes instruction cache, 16Kbytes data cache, and power management features.

Major features of the MPC8240 are as follows:

Peripheral logic

- **Memory interface**
  - High-bandwidth bus, 64-bit data bus, to SDRAM.
  - ECC Protected SDRAM
  - 16 Mbytes of ROM space (32Mbytes paged).
  - 8-bit ROM.
  - Write buffering for PCI and processor accesses.
- **PCI Interface**
  - 32-bit PCI interface operating at 33 MHz (66 MHz capable).
  - PCI 2.1-compatible.
  - Support for accesses to all PCI address spaces.
  - Selectable big- or little-endian operation.
  - Store gathering of processor-to-PCI write and PCI-to-memory write accesses.
  - PCI bus arbitration unit (five request/grant pairs).
- **Two-channel integrated DMA controller**
  - Supports direct mode or chaining mode (automatic linking of DMA transfers).
  - Supports scatter gathering read or write discontinuous memory.
  - Interrupt on completed segment, chain, and error.
  - Local-to-local memory.
  - PCI-to-PCI memory.
  - PCI-to-local memory.
  - Local-to-PCI memory.
- **Message unit**
  - Two doorbell registers.
  - Inbound and outbound messaging registers.
  - I 2 O message controller.
- **I 2 C controller with full master/slave support**
- **Embedded programmable interrupt controller (EPIC)**
  - Five hardware interrupts (IRQs) or 16 serial interrupts.
  - Four programmable timers.
- **Integrated PCI bus and SDRAM clock generation**
- **Programmable memory and PCI bus output drivers**
- **Debug features**
  - Memory attribute and PCI attribute signals.
  - Debug address signals.
  - MIV signal: Marks valid address and data bus cycles on the memory bus.
  - Error injection/capture on data path.
  - IEEE 1149.1 (JTAG)/test interface.

Processor core

- **High-performance, superscalar processor core**
  - Integer unit (IU).
  - Foating-point unit (FPU) (user enabled or disabled).
  - Load/store unit (LSU).
  - System register unit (SRU).
  - Branch processing unit (BPU).

- **16-Kbyte instruction cache**
- **16-Kbyte data cache**
- **Lockable L1 cache - *entire cache or on a per-way basis***
- Dynamic power management

4.7.10.1 Address Map

The MPC8240 in PCI host mode supports two address mapping configurations designated as address map A, and address map B. Address map A conforms to the PowerPC reference platform (PReP) specification. Address map B conforms to the PowerPC microprocessor common hardware reference platform (CHRP). Note that the support of map A is provided for backward compatibility only. It is strongly recommended that new designs use map B because map A may not be supported in future devices.

Address map B complies with the PowerPC microprocessor common hardware reference platform (CHRP). The address space of map B is divided into four areas: system memory, PCI memory, PCI I/O, and system ROM space. When configured for map B, the MPC8240 translates addresses across the internal peripheral logic bus and the external PCI bus as shown in Table 6.

Table 6. MPC8240 Address Map B

Processor Core Address Range				PCI Address Range	Definition
Hex		Decimal			
0000_0000	0009_FFFF	0	640K - 1	NO PCI CYCLE	System memory
000A_0000	000F_FFFF	640K	1M-1	000A_0000 – 000F_FFFF	Compatibility hole
0010_0000	3FFF_FFFF	1M	1G-1	NO PCI CYCLE	System memory
4000_0000	7FFF_FFFF	1G	2G-1	NO PCI CYCLE	Reserved
8000_0000	FCFF_FFFF	2G	4G-48M-1	8000_0000 – FCFF_FFFF	PCI memory
FD00_0000	FDFF_FFFF	4G-48M	4G-32M-1	0000_0000 – 00FF_FFFF	PCI/ISA memory
FE00_0000	FE7F_FFFF	4G-32M	4G-24M-1	0000_0000 – 007F_FFFF	PCI/ISA I/O
FE80_0000	FEBF_FFFF	4G-24M	4G-20M-1	0080_0000 – 00BF_FFFF	PCI I/O
FEC0_0000	FEDF_FFFF	4G-20M	4G-18M-1	CONFIG_ADDR	PCI configuration address
FEE0_0000	FEEF_FFFF	4G-18M	4G-17M-1	CONFIG_DATA	PCI configuration data
FEF0_0000	FEFF_FFFF	4G-17M	4G-16M-1	FEF0_0000 – FEFF_FFFF	PCI interrupt acknowledge
FF00_0000	FF7F_FFFF	4G-16M	4G-8M-1	FF00_0000 – FF7F_FFFF	32/64-bit FLASH/ROM (1)
FF80_0000	FFFF_FFFF	4G-8M	4G-1	FF80_0000 – FFFF_FFFF	8/32/64-bit FLASH/ROM (2)

Notes:
(1) This bank of FLASH is not used.
(2) This bank of FLASH is configured in 8-bit mode and is further broken down in Table 7.

Table 7. Port X Address Map

Bank Select	Processor Core Address Range		Definition
11111	FFE0_0000	FFEF_FFFF	Accesses Bank 0
11110 - 00001	FFE0_0000	FFEF_FFFF	Application code (1) (30 pages)
00000	FFE0_0000	FFEF_FFFF	Application/boot code (1), (2)
XXXX (3)	FFF0_0000	FFFF_CFFF	Application/boot code (2)
	FFFF_D000	FFFF_D000	Discrete input word 0
	FFFF_D001	FFFF_D001	Discrete input word 1
	FFFF_D002	FFFF_D002	Discrete output word 0
	FFFF_D003	FFFF_D003	Discrete output word 1
	FFFF_D004	FFFF_D004	Discrete output word 2
	FFFF_D010	FFFF_D010	IC (Pending interrupt)
	FFFF_D011	FFFF_D011	IC (Interrupt mask low)
	FFFF_D012	FFFF_D012	IC (Interrupt clear low)
	FFFF_D013	FFFF_D013	IC (Unmasked, pending low)
	FFFF_D014	FFFF_D014	IC (Interrupt input low)
	FFFF_D015	FFFF_D015	Unused (read FF)
	FFFF_D016	FFFF_D016	Unused (read FF)
	FFFF_D017	FFFF_D017	Unused (read FF)
	FFFF_D018	FFFF_D018	Unused (read FF)
	FFFF_D019	FFFF_D019	Unused (read FF)
	FFFF_D020	FFFF_D020	HA (Local HA register)
	FFFF_D021	FFFF_D021	HA (Node 0 HA register)
	FFFF_D022	FFFF_D022	HA (Node 1 HA register)
	FFFF_D023	FFFF_D023	HA (Node 2 HA register)
	FFFF_D024	FFFF_D024	HA (Node 3 HA register)
	FFFF_D025	FFFF_D025	HA (8240 HA register)
	FFFF_D026	FFFF_D026	HA (Software Fail)
	FFFF_D027	FFFF_D027	HA (Watchdog Strobe)
	FFFF_D028	FFFF_DFFF	4068 Bytes FLASH
	FFFF_E000	FFFF_FFFF	8K NOVRAM

Notes:
(1) Thirtyone 1Mbyte blocks of application memory residing at address FFE0_0000 – FFEF_FFFF selected by the FLASH page bits.
(2) 2Mbyte block available after reset.
(3) Always available

4.7.10.2 Register Description

Reference 10 describes the registers of the MPC8240.

4.7.10.3 Interrupt

The MPC8240 contains an embedded programmable interrupt controller (EPIC) device. The EPIC implements the necessary functions to provide a flexible and general-purpose interrupt controller solution. The EPIC pools hardware-

generated interrupts from many sources, both within the MPC8240 and externally, and delivers them to the processor core in a prioritized manner. The solution adopts the OpenPIC architecture (architecture developed jointly by AMD and Cyrix for SMP interrupt solutions) and implements the logic and programming structures according to that specification. The MPC8240's EPIC unit supports up to five external interrupts, four internal logic-driven interrupts and four timers with interrupts. See Reference 10 for a detailed description of the EPIC unit.

The five external interrupt inputs to the EPIC are wired to the external interrupt controller PLD.

4.7.10.4 MPC8240 Reset

The MPC8240 can be reset from three sources: a board level reset (RESET_0), JTAG controlled reset, or a failure in it's watchdog monitor. Any reset to the MPC8240 shall cause the discrete output registers to reset (low) state, this in turn, will cause all G4 nodes to enter the reset state.

4.7.10.5 Boot Procedure

After the release of reset to the MPC8240, it will begin executing code out of the FLASH memory. A reset will automatically set the FLASHSEL(4:0) bits to all zero's, therefore, the MPC8240's boot code must reside in bank 0. Once it's application code is copied to SDRAM, the MPC8240 can then sequence through the FLASH banks by setting the appropriate bits in the discrete output word. Application code for the G4 nodes resides in the remaining thirtyone banks of FLASH.

4.7.11 Bulk FLASH Memory

There are 32Mbytes of bulk FLASH memory, comprised of two Intel 28F128J3 StrataFLASH memory devices. The MPC8240's memory map limits the size of the 8-bit wide FLASH to 2Mbytes, this requires hardware to divide the FLASH into thirty-two 1Mbyte banks. Five software-controlled discretes allow switching between banks. Accesses to the 1Mbyte address range of FFE0_0000 through FFEF_FFFF will always access the first first block of FLASH, NOVRAM,Discrete I/O, HA registers, watchdog monitor, and the interrupt controller. Accesses to the 1Mbyte address range of FFF0_0000 through FFFF_FFFF will access a page of memory in the FLASH. The actual page is selected is based on the five FLASH select bits, driven by the Discrete Output word.

4.7.12 Real Time Clock

The PCF8563 is a CMOS real-time clock/calendar optimized for low power consumption. A programmable clock output, interrupt output and voltage-low detector are also provided. All addresses and data are transferred serially via a two-line bidirectional I 2 C-bus. Maximum bus speed is 400 kbits/s.

Real Time Clock Features:

- Provides year, month, day, weekday, hours, minutes and seconds
  (Based on an external 32.768 kHz quartz crystal)
- Century flag
- Wide operating supply voltage range: 1.0 to 5.5 V
- Low back-up current; typical 0.25 mA at VDD = 3.0 V and Tamb =2 °C
- 400 kHz two-wire I 2 C-bus interface (at VDD = 1.8 to 5.5 V)
- Programmable clock output for peripheral devices: 32.768 kHz, 1024 Hz, 32 Hz and 1 Hz
- Alarm and timer functions
- Voltage-low detector
- Integrated oscillator capacitor
- Internal power-on reset
- I 2 C-bus slave address: read A3H; write A2H
- Open drain interrupt pin

4.7.13 NonVolatile Memory

The MPC8240 will be equipped with 8Kx8 of non-volatile RAM for the storage of fault record data and configuration information. This function is implemented using a SIMTEK STK12C68S45 NOVRAM attached to the local bus

interface. The device's data bus is isolated from the local bus through an IDT IDTQS32244SO buffer. This buffer provides 3.3v to 5v translation.

4.7.14 Fault Status and Control Registers

The MPC8240 has access to five 8-bit status registers. One register represents its own status while the others represent that fault status of the other four G4 CPUs. Each register has the identical format as shown in Table 8:
These five registers grant the MPC8240 status information from each node on the board, without going through the Raceway fabric.

The MPC8240 will have one 8-bit Fault control register. The control register for each CPU will have the following format as shown in Table 9:

Bit	Name	Description
0	CHECKSTOP_OUT	Checkstop state of CPU (0 = CPU in checkstop)
1	WDM_FAULT	WDM failed (0 = WDM failed, set high after reset and valid service)
2	SOFTWARE_FAULT	Software fault detected (Set to 0 when a software exception was detected) **(R/W local)**
3	RESETREQ_IN	Wrap status of the local CPU's reset request
4	WDM_INIT	WDM failed in initial 2 second window (0 = WDM failed)
5	Software definable 0	Software definable 0
6	Software definable 1	Software definable 1
7	unused	unused

Table 8. Fault Status Register Format

Bit	Name	Description
0	RESETREQ_OUT_0	Request a reset event (0 => forces reset)
1	CHKSTOPOUT_0	Request that node 0 enter checkstop state (0 => request checkstop)
2	CHKSTOPOUT_1	Request that node 1 enter checkstop state (0 => request checkstop)
3	CHKSTOPOUT_2	Request that node 2 enter checkstop state (0 => request checkstop)
4	CHKSTOPOUT_3	Request that node 3 enter checkstop state (0 => request checkstop)
5	CHKSTOPOUT_8240	Request that the MPC8240 enter checkstop state (0 => request checkstop)
6	Software definable 0	Software definable 0
7	Software definable 1	Software definable 1

Table 9. Fault Control Register Definition

4.7.15 Majority Voter

There are two different functions controlled by majority voters. The first is local to each CPU, this voter controls the assertion of CHECKSTOP_IN to the CPU. The second voter is centralized to the board, it will control the master reset to the board. Both voters shall follow the same set of rules: The output will follow the majority of non-checkstopped CPUs. A 1-on-1 or 2-on-2 condition in either voter will result in a board level reset.

4.7.16 Discrete I/O

There are 16 discrete output signals directly controllable and readable by the MPC8240. The 16 discretes are divided up into two addressable 8-bit words. Writing to a discrete output register will cause the upper 8-bits of the data bus to be written to the discrete output latch. Reading a discrete output register will drive the 8-bit discrete output onto the upper 8-bits of the MPC8240's data bus. Table 10 defines the bits in the discrete output word.

There are 16 discrete input signals accessible by the MPC8240. Reads from the discrete input address space will latch the state of the signals, and return the latched state of the discretes to the MPC8240. Table 11 defines the bits in the discrete input word.

Table 10. Discrete Output Words

Word 2		
DH(0:7)	Signal	Description
0	ND0_FLASH_EN_1	Enable the CE ASIC's FLASH port when 1
1	ND1_FLASH_EN_1	Enable the CE ASIC's FLASH port when 1
2	ND2_FLASH_EN_1	Enable the CE ASIC's FLASH port when 1
3	ND3_FLASH_EN_1	Enable the CE ASIC's FLASH port when 1
4	Wrap 1	Wrap to discrete input
5		
6		
7		

Word 1		
DH(0:7)	Signal	Description
0	WRAP0	Wrap to Discrete Input
1	I2C_RESET_0	Reset the I2C serial bus when 0
2	SWLED	Software controlled LED
3	FLASHSEL4	Flash bank select address bit 4
4	FLASHSEL3	Flash bank select address bit 3
5	FLASHSEL2	Flash bank select address bit 2
6	FLASHSEL1	Flash bank select address bit 1
7	FLASHSEL0	Flash bank select address bit 0

Word 0		
DH(0:7)	Signal	Description
0	C_SRESET3_0	Issue a Soft Reset to cpu on Node 3 when 0
1	C_PRESET3_0	Reset PCE133 ASIC Node 3 when 0
2	C_SRESET2_0	Issue a Soft Reset to cpu on Node 2 when 0
3	C_PRESET2_0	Reset PCE133 ASIC Node 2 when 0
4	C_SRESET1_0	Issue a Soft Reset to cpu on Node 1 when 0
5	C_PRESET1_0	Reset PCE133 ASIC Node 1 when 0
6	C_SRESET0_0	Issue a Soft Reset to cpu on Node 0 when 0
7	C_PRESET0_0	Reset PCE133 ASIC Node 0 when 0

Table 11. Discrete Input Words

Word 1		
DH(0:7)	Signal	Description
0	WRAP1	Wrap from discrete output word
1	TBD	
2	V3.3_FAIL_0	Latched status of power supply since last reset
3	V2.5_FAIL_0	Latched status of power supply since last reset
4	VCORE1_FAIL_0	Latched status of power supply since last reset
5	VCORE0_FAIL_0	Latched status of power supply since last reset
6	RIOR_CNF_DONE_1	RIO/RACE++ FPGA configuration complete
7	PXB0_CNF_DONE_1	PXB++ FPGA configuration complete

		Word 0
DH(0:7)	Signal	Description
0	WRAP0	Wrap from discrete output word
1	WDMSTATUS	MPC8240's watchdog monitor status (0 = failed)
2	NPORESET_1	Not a power on reset when high
3		
4		
5		
6		
7		

4.7.17 Interrupt Controller

The MPC8240 interfaces with an 8-input interrupt controller external from MPC8240 itself. The interrupt inputs are wired, through the controller to interrupt zero of the MPC8240 external interrupt inputs. The remaining four MPC8240 interrupt inputs are unused.

The Interrupt Controller comprises the following five 8-bit registers;

Pending Register - A low bit indicates a falling edge was detected on that interrupt (read only)
Clear Register - Setting a bit low will clear the corresponding latched interrupt (write only)
Mask Register – Setting a bit low will mask the pending interrupt from generating an MPC8240 interrupt
Unmasked Pending Register – A low bit indicates a pending interrupt that is not masked out
Interrupt State Register – indicates the actual logic level of each interrupt input pin

4.7.17.1 Interrupt Controller Operation

Table 12 lists the interrupt input sources and their bit positions within each of the six registers. A falling edge on an interrupt input will set the appropriate bit in the pending register low. The pending register is gated with the mask register and any unmasked pending interrupts will activate the interrupt output signal to the MPC8240's external interrupt input pin. Software will then read the unmasked pending register to determine which interrupt(s) caused the exception. Software can then clear the interrupt(s) by writing a zero to the corresponding bit in the clear register. If multiple interrupts are pending, the software has the option of either servicing all pending interrupts at once and then clearing the pending register or servicing the highest priority interrupt (software priority scheme) and the clearing that single interrupt. If more interrupts are still latched, the interrupt controller will generate a second interrupt to the MPC8240 for software to service. This will continue until all interrupts have been serviced. An interrupt that is masked will show up in the pending register but not in the unmasked pending register and will not generate an MPC8240 interrupt. If the mask is then cleared, that pending interrupt will flow through the unmasked pending register and generate an MPC8240 interrupt.

Table 12. Interrupt Controller Inputs

Bit	Signal	Description
0	SWFAIL_0	8240 Software Controlled Fail Discrete
1	RTC_INT_0	Real time clock event
2	NODE0_FAIL_0	WDFAIL_0 or IWDFAIL_0 or SWFAIL_0 active
3	NODE1_FAIL_0	WDFAIL_0 or IWDFAIL_0 or SWFAIL_0 active
4	NODE2_FAIL_0	WDFAIL_0 or IWDFAIL_0 or SWFAIL_0 active
5	NODE3_FAIL_0	WDFAIL_0 or IWDFAIL_0 or SWFAIL_0 active
6	PCI_INT_0	PCI interrupt
7	XB_SYS_ERR_0	XBAR internal error

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ❑ BLACK BORDERS
- ❑ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ❑ FADED TEXT OR DRAWING
- ❑ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ❑ SKEWED/SLANTED IMAGES
- ❑ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ❑ GRAY SCALE DOCUMENTS
- ❑ LINES OR MARKS ON ORIGINAL DOCUMENT
- ❑ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ❑ OTHER: ______________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.

4.7.18 Configuration Jumpers

J18-1 – J18-2, the watchdog monitor mask, when installed, will mask all watchdog failures.

J18-3 – J18-4, the serial EEPROM's write enable jumper, when installed, enables modification of the serial EEPROMs.

J18-5 – J18-6, the flash write-protect jumper, when installed, prevents modification of any flash memory location.

J18-7 – J18-8, the PXB0 use PROM jumper, when installed will enable the PXB0's serial configuration PROM.

4.7.19 LEDs

There are nine LEDs, visible at the top of the board.

LD1 is a software controlled LED
LD2 is a software controlled LED
LD3 is the Node 0 watchdog fail LED
LD4 is the Node 1 watchdog fail LED
LD5 is the Node 2 watchdog fail LED
LD6 is the Node 3 watchdog fail LED
LD7 is the MPC8240 watchdog fail LED
LD8 indicates the state of the board level reset
LD9 indicates a XBAR system error.

There are an additional two LEDs on the Ethernet connector for Ethernet status (located on the Ethernet connector).

4.7.20 Power Supply

The MCW-1a board requires 3.3V, 2.5V, and 1.8V. There are two 1.8V supplies, each drives the core voltage for two cpus. To provide power to the MCW-1a, the three voltages must have separate switching supplies, and proper power sequencing to the device must be provided. All three voltages are converted from 5.0V. The power to the daughter card is provided directly from the modem board.

4.7.20.1 MPC7400 Core Power Supply

There are two core voltage power supplies, each one is dedicated to two MPC7400 PPC cores. The core voltage can be in the 2.2V to 1.5V range. This power supply is rated at 12A in the range from 2.2V to 1.5V.

4.7.20.2 Main 3.3V Power Supply

A 3.3V power supply is used to provide power to the SBSRAM core, SDRAM, SCSI, PXB++, and XBAR++ PCE133 I/O. This power supply is rated at TBD Amp.

4.7.20.3 Core and I/O 2.5V Power Supply

A 2.5V power supply is used to provide power to the PCE133 and can also power the PXB++ FPGA core. The MPC7400 processor bus can run at 2.5V signaling. The MPC7400 L2 bus can operate at 2.5V signaling. This 2.5V power supply is rated at TBD Amp.

4.7.20.4 ASICs Power Supplies Tolerance Requirements

SBSRAM VDD = 3.3V+0.165V/-0.165V power supply
SBSRAM VDDQ = 3.3V+0.165V/-0.165V for 3.3V I/O or 2.5V+0.4V/-0.125V for 2.5V I/O
SDRAM VDD= 3.3V+0.3V/-0.3V power supply
XBAR++ VDD= 3.3V+0.3V/-0.3V power supply
PCE133 VDD= 2.5V+?V/-?V power supply
PCE133 VDD33= 3.3V+?V/-?V power supply

4.7.20.5 Power Supply Voltage Sequencing

The power sequencing is very important in multivoltage digital boards. It is necessary for long-term reliability. The right power supply sequencing can be accomplished by using *power_good* and *inhibit* signals. To provide fail-safe operation of the device, power should be supplied so that if the core supply fails during operation, the I/O supply is shut down as well.

The general rule is to ramp all power supplies up and down at the same time. This is shown in Figure 5. In reality, ramp up and down depend on multiple factors: power supply, total board capacities that need to be charged, power supply load, and so on. Figure 6 shows ideal worst-case sequencing for ramp up and down that is performed by the protection sequencing circuits shown in Figure 7. This circuit keeps the voltage difference within the required range.
The MPC7400 requires the core supply to not exceed the I/O supply by more than 0.4 volts at all times. Also, the I/O supply must not exceed the core supply by more than 2 volts.

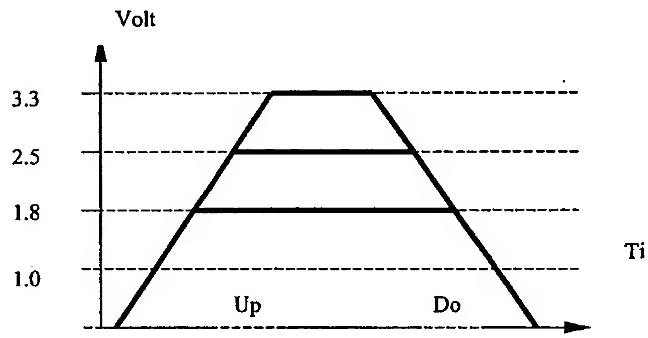


Figure 5. IDEAL POWER SUPPLY SEQUENCING

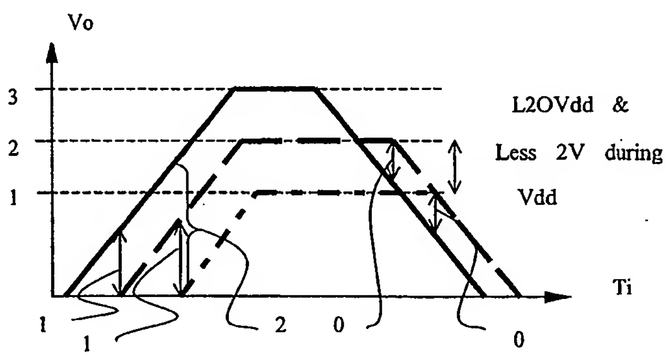
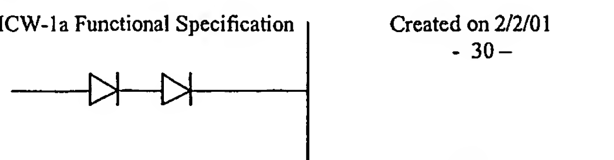


Figure 6. REAL POWER SUPPLY SEQUENCING

3.3V 2.5V 1.8V_1



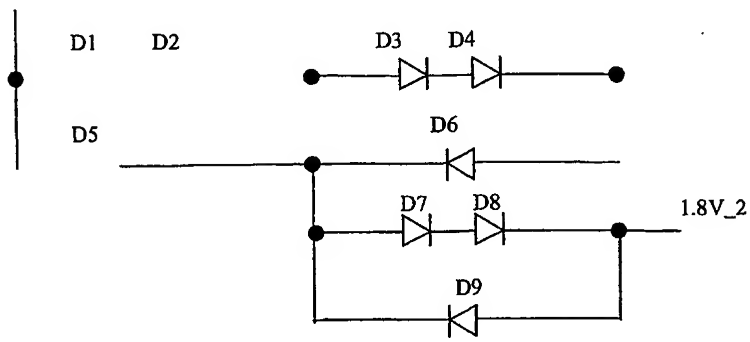


Figure 7. VOLTAGE SEQUENCING CIRCUITS

0.7V voltage drops across one diode.

During power up sequencing:
D1 and D2 provide the ramp up voltage for the 2.5V power supply as soon as the 3.3V power supply reaches 1.4V.
D3 and D4 provide the ramp up voltage for the 1.8V_1 power supply as soon as the 2.5V power supply reaches 1.4V.
D7 and D8 provide the ramp up voltage for the 1.8V_2 power supply as soon as the 2.5V power supply reaches 1.4V.

During power down sequencing:
D5 provides the ramp down for the 2.5V power supply as soon as the 3.3V power supply reaches 1.8V.
D6 provides the ramp down for the 1.8V_1 power supply as soon as the 2.5V power supply reaches 1.1V.
D9 provides the ramp down for the 1.8V_2 power supply as soon as the 2.5V power supply reaches 1.1V.

The 3.3V power supply is connected to the VCC3P3 power plane.
The 2.5V power supply is connected to the VCC2P5 power plane.
The 1.8V_1 power supply is connected to the VCC1P8_1 power plane.
The 1.8V_2 power supply is connected to the VCC1P8_2 power plane.

4.7.20.6 Power Supply Monitoring

A PLD is used to monitor the voltage status signals from the onboard supplies. It is powered up from +5V and monitors +3.3V, +2.5V, 1.8V_1 and +1.8V_2. This circuit monitors the *power_good* signals from each supply. In the case of a power failure in one or more supplies, the PLD will issue a restart to all supplies and a board level reset to the daughter card. A latched power status signal will be available from each supply as part of the discrete input word. The latched discrete shall indicate any power fault condition since the last off-board reset condition.

5 ELECTRICAL INTERFACE

5.1.1 Power Consumption

Table 13. MCW-1a CN Power Consumption

Description	Qty	Total Typ. Power	Total Max. Pwr.
CE ASIC	1	1W	1.5W
SDRAM	5	3W	3.5W
SBSRAM	2	1.2W	2.5W
G4	1	8W	12W
Oscillator	1	0.1W	0.1W
PLD	1	0.15W	0.2W

TBD

Table 14. MCW-1a Power Consumption

TBD

5.1.2 I/O

5.1.2.1 Over-the-Top RACEway++ Interlink
See Appendix A for the over-the-top RACEway++ interlink connector pinout.

5.1.2.2 PCI 32-Bit Modem Connector
See Appendix B for the PCI 32-bit modem connector pinout.

5.1.2.3 Ethernet 10/100BT
See Appendix C for the Ethernet 10/100 BT connector pinout.

5.1.2.4 PPC Debugger
See Appendix D for the PPC Debugger connector pinout.

6 MECHANICAL

6.1.1 Packaging

The MCW-1is a dual-side PCB assembly. The board is designed to be used in a custom system. The MCW-1PCB is TBD thick and TBD layers.

6.1.2 Physical Constraint

The PCB board must comply with the Motorola daughter card form factor.

7 ENVIRONMENTAL

7.1.1 Temperature & Air Flow

Operating temperature: TBD
Storage temperature: TBD

7.1.2 Humidity

TBD

7.1.3 Operating Altitude

TBD

7.1.4 Shock & Vibration

TBD

7.1.5 Compliance

TBD

7.1.6 Reliability

TBD

8 SWITCHES & JUMPERS

8.1 J22 Jumper

Provisional Hotswap switch interface for the PXB0.

J22 Ref. Des.	Jumper Function
1 – 2	PXB0_HS_HNDL_SW high
2 – 3	PXB0_HS_HNDL_SW low

8.2 J11 Jumper

Raceway clock master selection

J11 Ref. Des.	Jumper Function
1 – 2 (open)	MCW-1A Master
1 – 2 (shorted)	MCW-1A Slave

8.3 J10 Jumper
F1 Raceway XBREQI – XBREQO crossover.

J10 Ref. Des.	Jumper Function
3 – 4, 5 – 6	Straight through
1 – 2, 7 - 8	Crossover

8.4 J4 Jumper
F2 Raceway XBREQI – XBREQO crossover.

J4 Ref. Des.	Jumper Function
3 – 4, 5 – 6	Straight through
1 – 2, 7 - 8	Crossover

8.5 J3 Jumper
F2 Raceway CBL_CLK_O – CBL_CLK_I crossover.

J3 Ref. Des.	Jumper Function
3 – 4, 5 – 6	Straight through
1 – 2, 7 - 8	Crossover

8.6 J9 Jumper
F1 Raceway CBL_CLK_O – CBL_CLK_I crossover.

J9 Ref. Des.	Jumper Function
3 – 4, 5 – 6	Straight through
1 – 2, 7 - 8	Crossover

8.7 J18 Jumper
Miscellaneous control

J18 Ref. Des.	Jumper Function
1 – 2	WDM fail disable
3 – 4	Serial PROM write enable
5 – 6	FLASH write enable
7 – 8	PXB0 use configuration PROM
9– 10	Unused

8.8 J21 Jumper
Master clock source selector

J21 Ref. Des.	Jumper Function
1 – 2	F1 cable port master
3 – 4	F2 cable port master
Both closed	MCW-1A master
Both open	MCW-1A master

9 TESTABILITY

9.1 JTAG Test Scan

The MPC7400, MPC8240, PCI-PCI bridge, PCE133 ASIC, PXB++ ASIC, XBAR++ ASIC, and the Ethernet controller provide support for the IEEE Standard 1149.1 test port (JTAG). Refer to the individual component specifications to obtain their JTAG test access port (TAP) descriptions.

The MCW-1a board contains several JTAG scan chains. They provide access to the JTAG test port on the MPC7400s, MPC8240, L2 caches, XBAR++, PCE133s, Ethernet, PCI-PCI bridge, and the PXB devices. The scan chain is defined as;
Chain 1 -> MPC7400_1
Chain 2 -> MPC7400_2
Chain 3 -> MPC7400_3
Chain 4 -> MPC7400_3
Chain 5 -> MPC8240
Chain 6 -> RESET_PLD, PCEFIX1_PLD, NODE0_HA_PLD, NODE1_HA_PLD, PCEFIX2_PLD, NODE2_HA_PLD, NODE3_HA_PLD, 8240_DECODE_PLD, VOTER_SYNC_PLD, 8240_HA_PLD, PXB_PROM, L2 Cache_1, PCE133_1, L2 Cache_2, PCE133_2, XBAR, L2_Cache_3, PCE133_3, L2 Cache_4, PCE133_4, PXB++, PCI-PCI Bridge, Ethernet

The scan path is accessible via connector J16. The enable for the scan chain buffer is controlled by jumper J20.

The RACEway++ interlink external connectors will be tested with external loop-back connectors.

Note: Both the RACEway++ clock (66 MHz) and the PCI clock (33 MHz) must be running to allow the scan path in the PXB to function properly.

10 RACEway++ Over-the-Top Connector Pinout

Table 15. RACEway++ F1 Cable Mode Connector Pinout J-1

Pin	Signal	Pin	Signal
A1	GND	B1	CLK_X_JX1_IO
A2	GND	B2	JX1_CBL_CLK_IO
A3	GND	B3	JX1_XBREQ_I
A4	GND	B4	JX1_XBREQ_O
A5	GND	B5	JX1_XBSTROBIO
A6	GND	B6	JX1_XBRPLYIO
A7	GND	B7	JX1_XBRDCONIO
A8	GND	B8	JX1_XBIO00
A9	GND	B9	JX1_XBIO01
A10	GND	B10	JX1_XBIO02
A11	GND	B11	JX1_XBIO03
A12	GND	B12	JX1_XBIO04
A13	GND	B13	JX1_XBIO05
A14	GND	B14	JX1_XBIO06
A15	GND	B15	JX1_XBIO07
A16	GND	B16	JX1_XBIO08
A17	GND	B17	JX1_XBIO09
A18	GND	B18	JX1_XBIO10

A19	GND	B19	JX1_XBIO11
A20	GND	B20	JX1_XBIO12
A21	GND	B21	JX1_XBIO13
A22	GND	B22	JX1_XBIO14
A23	GND	B23	JX1_XBIO15
A24	GND	B24	JX1_XBIO16
A25	GND	B25	JX1_XBIO17
A26	GND	B26	JX1_XBIO18
A27	GND	B27	JX1_XBIO19
A28	GND	B28	JX1_XBIO20
A29	GND	B29	JX1_XBIO21
A30	GND	B30	JX1_XBIO22
A31	GND	B31	JX1_XBIO23
A32	GND	B32	JX1_XBIO24
A33	GND	B33	JX1_XBIO25
A34	GND	B34	JX1_XBIO26
A35	GND	B35	JX1_XBIO27
A36	GND	B36	JX1_XBIO28
A37	GND	B37	JX1_XBIO29
A38	GND	B38	JX1_XBIO30
A39	JX1_XBPAR	B39	JX1_XBIO31
A40	+3.3V	B40	R_RST_JX

Table 16. RACEway++ F2 Cable Mode Connector Pinout J-2

Pin	Signal	Pin	Signal
A1	GND	B1	CLK_X_JX2_IO
A2	GND	B2	JX2_CBL_CLK_IO
A3	GND	B3	JX2_XBREQ_I
A4	GND	B4	JX2_XBREQ_O
A5	GND	B5	JX2_XBSTROBIO
A6	GND	B6	JX2_XBRPLYIO
A7	GND	B7	JX2_XBRDCONIO
A8	GND	B8	JX2_XBIO00
A9	GND	B9	JX2_XBIO01
A10	GND	B10	JX2_XBIO02
A11	GND	B11	JX2_XBIO03
A12	GND	B12	JX2_XBIO04
A13	GND	B13	JX2_XBIO05
A14	GND	B14	JX2_XBIO06
A15	GND	B15	JX2_XBIO07
A16	GND	B16	JX2_XBIO08
A17	GND	B17	JX2_XBIO09
A18	GND	B18	JX2_XBIO10
A19	GND	B19	JX2_XBIO11

A20	GND	B20	JX2_XBIO12
A21	GND	B21	JX2_XBIO13
A22	GND	B22	JX2_XBIO14
A23	GND	B23	JX2_XBIO15
A24	GND	B24	JX2_XBIO16
A25	GND	B25	JX2_XBIO17
A26	GND	B26	JX2_XBIO18
A27	GND	B27	JX2_XBIO19
A28	GND	B28	JX2_XBIO20
A29	GND	B29	JX2_XBIO21
A30	GND	B30	JX2_XBIO22
A31	GND	B31	JX2_XBIO23
A32	GND	B32	JX2_XBIO24
A33	GND	B33	JX2_XBIO25
A34	GND	B34	JX2_XBIO26
A35	GND	B35	JX2_XBIO27
A36	GND	B36	JX2_XBIO28
A37	GND	B37	JX2_XBIO29
A38	GND	B38	JX2_XBIO30
A39	JX2_XBPAR	B39	JX2_XBIO31
A40	+3.3V	B40	R_RST_JX

11 Modem Board Connector Pinout

Table 17. Modem Board Connector Pin Assignments

J29			
Pin	Signal	Signal	Pin
1	5V	PMC_AD0	2
3	5V	PMC_AD1	4
5	5V	PMC_AD2	6
7	5V	PMC_AD3	8
9	PCI_RST_0	PMC_AD4	10
11	GND	PMC_AD5	12
13	GND	PMC_AD6	14
15	PMC_IDSEL_1	PMC_AD7	16
17	5V	PMC_AD8	18
19	5V	PMC_AD9	20
21	PMC_TRDY_0	PMC_AD10	22
23	GND	PMC_AD11	24
25	GND	PMC_AD12	26
27	PMC_STOP_0	PMC_AD13	28
29	5V	PMC_AD14	30
31	5V	PMC_AD15	32
33	PMC_PERR_0	PMC_AD16	34
35	GND	PMC_AD17	36
37	GND	PMC_AD18	38
39	PMC_SERR_0	PMC_AD19	40
41	5V	PMC_AD20	42
43	5V	PMC_AD21	44
45	CLK_PMC	PMC_AD22	46
47	GND	PMC_AD23	48
49	GND	PMC_AD24	50
51	PMC_C_BE0	PMC_AD25	52
53	PMC_C_BE1	PMC_AD26	54
55	5V	PMC_AD27	56
57	5V	PMC_AD28	58
59	PMC_C_BE2	PMC_AD29	60
61	PMC_C_BE3	PMC_AD30	62
63	GND	PMC_AD31	64
65	GND	5V	66

67	GND	PMC_FRAME_0	68
69	PMC_INTA_0	GND	70
71	GND	PMC_IRDY_0	72
73	GND	5V	74
75	PMC_GNT_0	PMC_DEVSEL_0	76
77	5V	PMC_LOCK_0	78
79	PMC_REQ_0	PMC_PAR	80

12 Processor JTAG Connector Pinout

The JTAG connectors are unique to each processor. Table 18 shows the generic signal names on each connector pin, the actual names will have each processor's extension appended to the generic signal name.

Table 18. JTAG Jx Connectors Pin Assignments

Jx-	SIGNAL	Jx-	SIGNAL
1	TDO	2	QACKN
3	TDI	4	TRSTN
5	HALTEDN	6	3.3V
7	TCK	8	CKSTOP_INN
9	TMS	10	N.C.
11	SRESETN	12	N.C.
13	HRESETN	14	<<key>>
15	CKSTOP_OUTN	16	GND

13 Non-Processor JTAG Connector Pinout

The non-processor JTAG connector ties together all the remaining JTAG capable devices together. Table 18 shows the signal names on each connector pin. The connector is designed to only include the programmable PLDs and PROM when the program cable is installed, or the entire chain when the Boundary scan test connector is installed.

Table 19. JTAG J16 Connectors Pin Assignments

J16-	Signal	Description
1	TMS_JTAG	JTAG Test Mode Select
2	TDI_JTAG	JTAG Test Data In
3	TDO_JTAG	Boundary Scan Test Data Out
4	TESTN	Driven low when connector inserted
5	TCK_JTAG	JTAG Test Clock
6	GND	Ground on module
7	PXB_CNF_TDO	TDO from end of PLD chain
8	TDI_NDO	TDI into non-PLD Chain
9	+5V	+5V Power on Module
10	TEST	Driven high when connector inserted

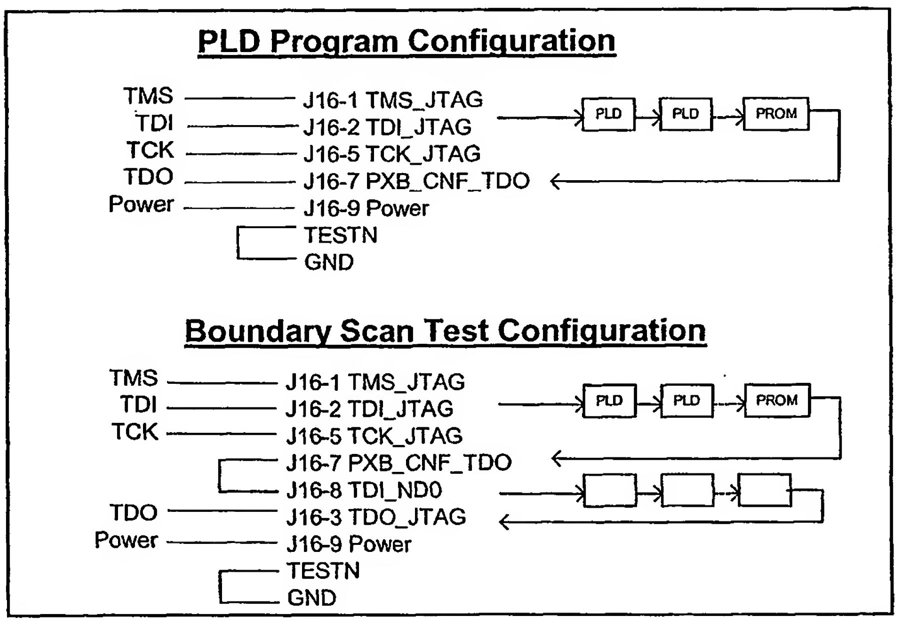


Figure 8. JTAG CONNECTOR CONFIGURATION OPTIONS

14 Design Notes

14.1 MPC7400 and Nitro Bus Signaling Voltage Support

	1.8V	2.5V	3.3V
MPC7400 V 60x	Yes	Yes	Yes
MPC7400 V L2	Yes	Yes	Yes
Nitro V 60x	Yes	Yes	No
Nitro V L2	Yes	Yes	No
PCE133 V 60x	No	Yes	No
SBSRAM Vi/o	No	Yes	Yes

14.2 Bypass Capacitors Selection

(Based on App. Note from Micron TN-00-06)

Vcore = 3.3V +/- 0.165V, which is 5%
Vi/o = 2.5V +/- 0.125V, which is 5%

When the SBSRAMs are driving 21pf load from 0V to 2.5V with 1ns edges, the transient current is:

I = (C * dV)/dt = (30pf*2.5V)/1ns = 75ma per one I/O pin.

For 36 I/O, 36*75ma = 2.7A in 1ns time interval.

The SyncBurst SRAM has a VDD tolerance of 3.3V +/-0.165V. Considering some droop from the power bus and a switching time of 1 ns, and allowing a maximum voltage dip (DV) on the SRAM of -0.05V, the choice of bypass capacitor becomes:

C = (I * dt)/dV = (2.7A * 1)/0.05 = 54nF per one SBSRAM.

Choosing 6 x 10nf allows some margin.
It is better to use reverse ratio capacitors 0508, 0406, or 0204.
The low ESR is also very important.
Temperature stable dielectric as X7R.
From Vishay VJ0402 style X7R.

14.3 Tantalum Capacitors Selection

Ultra-low ESR tantalum capacitors T510 are used in the switching power supply, besides several bulk storage capacitors distributed around the PCB that feed Vcore and Vi/o plains, to enable quick recharging of the bypass chip capacitors. The number of the bulk-storage tantalum capacitors depends on the power supply response time characteristic.

The MPC7400 can go from nap mode to full-on mode power within two cycles.

I core = (10W – 2W) /1.8V = 4.5A
dt = 10μs

C = (I * dt)/dV = (4.5A * 10μs) / 0.05V = 900μF

Computer Systems, Inc.
MERCURY

MEMORANDUM

TO	
FROM	Alden Fuchs
ABOUT	**Preliminary Framework interface**
	Memorandum # AF-4
VERSION	V0.2
DATE	8 December, 2000
COPIES TO	

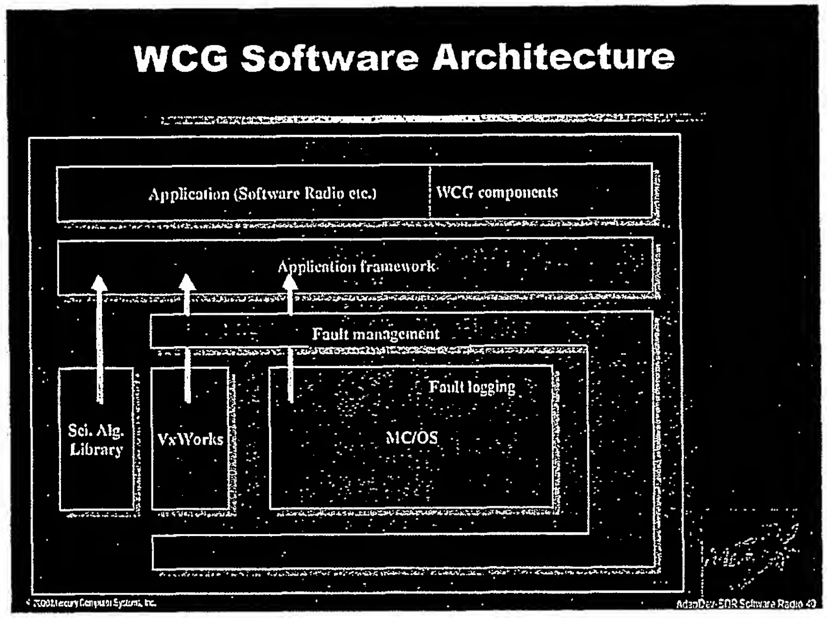
DISTRIBUTION

MEMO AF-4
Prototype Framework V0.1

MEMO AF-4
Prototype Framework V0.1

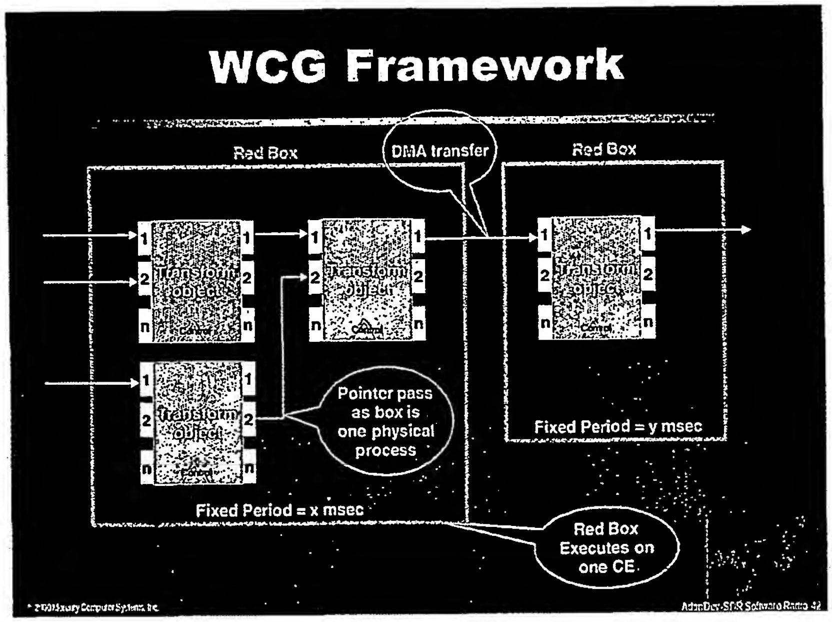
1. Introduction

This is a very brief description of the prototype framework and how to use it. The purpose of this memo is to describe the software interfaces from within a transform object.



The above figure depicts the software architecture, and the transform object is a part of the Application that is managed by the Application framework.

MEMO AF-4
Prototype Framework V0.1



1.1. Transform Object

The transform object is the basic building block and can be like a Turbo-coder, QAM modulator etc.

1.2. Red-Box

The red-box collects transform objects into a logical grouping that describes all of the processing that will be carried out on a single CPU.. (Note for reasons of non real-time operation eg simulation collections of red-boxes can be on a single CPU).

MEMO AF-4
Prototype Framework V0.1

2. Transform Object Sample

2.1. Include the following files to define the interface, and variables required..

#include “mc_error.h”
#include “mcw1.h”
#include “dx_dma.h”
#include “dx_dma_var.h”

2.1.1. The contents of dx_dma_var.h:

```
int my_logical_ce;
CONFIG_data *ptr_config_base;
CONFIG_data *ptr_cur_config;
CONFIG_data *ptr_tmp_config;

int active_in_ce[(MAX_CE+1) * MAX_CHAN];
int active_in_ch[(MAX_CE+1) * MAX_CHAN];
int active_in_buf_size[(MAX_CE+1) * MAX_CHAN];
char *active_in_buf[(MAX_CE+1) * MAX_CHAN];
int active_in_index;

int active_out_ce[(MAX_CE+1) * MAX_CHAN];
int active_out_ch[(MAX_CE+1) * MAX_CHAN];
int active_out_buf_size[(MAX_CE+1) * MAX_CHAN];
char *active_out_buf[(MAX_CE+1) * MAX_CHAN];
int active_out_index;

#define  dma_send_pin(pin) \
    dma_send(               \
            my_logical_ce,    \
            active_out_ce[pin], \
            active_out_ch[pin], \
            (char **)&active_out_buf[pin] \
            )

#define dma_rec_pin(pin) \
    dma_rec(               \
           active_in_ce[pin], \
           my_logical_ce,     \
           active_in_ch[pin], \
           (char **)&active_in_buf[pin] \
           )
```

2.2. Initialize the interface

// get config SMB

```
dma_all_init(
  my_logical_ce,
  active_in_ce,
  active_in_ch,
  active_in_buf_size,
```

MEMO AF-4
Prototype Framework V0.1

```
    active_in_buf,
    (int *)&active_in_index,
    active_out_ce,
    active_out_ch,
    active_out_buf_size,
    active_out_buf,
    (int *)&active_out_index,
    (CONFIG_data **)&ptr_config_base
              );
  ptr_cur_config = &ptr_config_base[my_logical_ce];
#ifdef debug_print
        printf("Vir CE %i, module name is %s\n",
my_logical_ce,ptr_cur_config->module_name);
#endif
  ptr_cur_config->state = STATE_RDY; /* all init done now ready */
//wait for rx to be ready
ptr_tmp_config = &ptr_config_base[active_out_ce[0]];
while (ptr_tmp_config->state != STATE_RDY) //need reciver to be ready
                                        sched_yield();
//wait for tx to be ready
ptr_tmp_config = &ptr_config_base[active_in_ce[0]];
while (ptr_tmp_config->state != STATE_RDY) //need reciver to be ready
                                        sched_yield();
#ifdef debug_print
  printf("\nCE %i, Virtual CE %i, Starting\n",(int)ce_getid(),my_logical_ce);
#endif
```

2.3. Receive input

Receive input data if required, input pins can be left unused.

2.3.1. An Example of the receiving of data from input pin 0

```
    /* get data from other CE */
    rc = dma_rec_pin(0);
    ERROR_MCW1(rc);
```

<u>OR</u>

```
    rc = dma_rec(
        active_in_ce[0],
        my_logical_ce,
        active_in_ch[0],
        (char **)&active_in_buf[0]
        );
    ERROR_MCW1(rc);
```

MEMO AF-4
Prototype Framework V0.1

The data is available in the active_in_buf pointer,, note this always points to the next available input buffer in the case of multi-buffering,, at a later date the size of input chunk and offset will be provided so that a FIFO like structure can be used.

2.4. Send Output

Send output data if required, output pins can be left unused.

2.4.1. An Example of the sending data on output pin 0

```
/* send data to other CE */
rc = dma_send_pin(0);
ERROR_MCW1(rc);
```

OR

```
rc = (long)dma_send(
  my_logical_ce,
  active_out_ce[0],
  active_out_ch[0],
  (char **)&active_out_buf[0]
  );
ERROR_MCW1(rc);
```

The data in the active_out_buf pointer will be sent, on return this always points to the next available output buffer in the case of multi-buffering. At a later date the size of output chunk and offset will be provided so that a FIFO like structure can be used.

MEMO AF-4
Prototype Framework V0.1

3. Transforms for WCDM Simulation:

3.1. handset (one of n):

This object has two input pins and one output pin. It performs the:

1. Generate transport channel
2. MUX and channel coding
3. Generate TX waveform
4. Simulate RX system for Power control etc.
5. Outputs to the chan model

3.1.1. input pins:

3.1.1.1. power_control pin 0 :
Input to this pin is from output pin 0 of the rake block and is the slot power control.

3.1.1.2. next_chunk pin 1:
Input to this pin is from output pin 1 of the BER block and is the send next n symbols for processing e.g. 2 symbols, or a slot etc.

3.1.1.3. next_chunk pin 1:
Optional input pin, used to provide external ie outside of the Generate traffic channel bits, access to the raw data input ie if we did a codec the output of the codec would go into this block.

3.1.2. Output pins:

3.1.2.1. signal_out pin 0 :
This pin goes to one input pin of the chan object group.

3.1.2.2. raw_bits pin 1:
This pin has the raw data bits as encoded into the Data channel so that the BER, BLER calculations can be done.

3.2. Chan (set of one to m objects):

In this group of objects, each has; two to n input pins; and one output pin each. They collectively perform the:

1. Channel model for each of the inputs except the carry pin
2. Sums the local signals, and adds the carry input pin
3. Outputs to the front_end object to send same data to all rake inputs

3.2.1. Input pins:

3.2.1.1. sum_in pin 0 :
Input to this pin is from output pin 0 of other channel object, currently a dummy input is required on this pin for the process to fire (needs more thought ie a special first chan??).

3.2.1.2. signal_in pin 1 to n:
Input to this pin is from output pin 0 of the handset block.

MEMO AF-4
Prototype Framework V0.1

3.2.2. Output pins:

3.2.2.1. signal_out pin 0 :
This pin goes to input pin 0 of the broadcast object.

3.3. front_end (one object):

In this object, each has; one input pin; and one output pin. It performs the:

1. ~~Adds the multiple antenna, and other~~Receiver distortions and noise
2. Simulate RX system (AGC, A/D, multiple antennas) etc.
3. Outputs to the broadcast object to send same data to all rake inputs

Multiple antennas should be treated as separate data streams. The rake receiver will process them independently, until the MRC stage.

3.3.1. Input pins:

3.3.1.1. signal_in pin 0 :
Input to this pin is from output pin 0 of the last channel object.

3.3.2. Output pins:

3.3.2.1. signal_out pin 0 to n:
This pin goes to input pin 0 of the broadcast objects.

3.4. broadcast (set of one to k objects):

This object is required to simulate broadcast, until the simple framework supports this feature, we need this object.

Each object in the group has one input pin and one to n output pins. They collectively perform the:

1. Takes one input and copies it to all of the output pins un-modified
2. Outputs same data to all rake input 0 pins.

3.4.1. Input pins:

3.4.1.1. signal_in pin 0 :
Input to this pin is from output pin 0 of the front_end object.

3.4.2. Output pins:

3.4.2.1. signal_out pin 0 to n:
This pin goes to input pin 0 of the rake objects.

3.5. Rake (one of n):

This object has one input pin and two output pins. It performs the:

1. ~~AGC,~~ AFC
2. Initial signal acquisition and s~~S~~earcher receiver~~RX~~
3. Multiple finger receivers~~Rx~~

MEMO AF-4
Prototype Framework V0.1

4. Channel estimation, MRC etc.
5. Final data channel despreading.
~~5.~~6. Outputs to:
   - MUD group of processes
   - Soft-decision symbol processing (FEC decoding and demultiplexing (25.212)

3.5.1. Input pins:

3.5.1.1. signal_in pin 0 :
This is the data from the broadcast set, and carries the signals of all the handsets, and noise etc.

3.5.2. Output pins:

3.5.2.1. power_control pin 0 :
This is the slot power control to be sent back to the handset.

3.5.2.2. signal_out pin 0 :
This pin goes to one input pin of the MUX object group.

3.6. MUX (set of one to L objects):

This object is required to gather and package information from the 1 to n rake objects. The inputs are placed into packets(???) or into arrays (???) To Be Determined (TBD). This object should be morphed into the best approximation of the packaging to be provided by a targeted modem.

Each object in the group has one to n input pins and one output pin. They collectively perform the:

1. Package rake information into simulated modem sourced data.
2. Outputs to MUD input 0 pin (for now until MUD integration there will be a dummy placeholder block).

3.6.1. Input pins:

3.6.1.1. signal_in pin 0 to L:
Input to this pin is from output pin 1 of the a rake object, or another MUX objects output pin 0 .

3.6.2. Output pins:

3.6.2.1. signal_out pin 0 :
This pin goes to input pin 0 of the rake objects.

3.7. MUD (one object for now):

This object is required to place hold until a real mud is implemented.

MUD has one input pin and one output pin.

1. Passes through data and formats it for the BER block
2. Outputs to BER input 0 pin.

MEMO AF-4
Prototype Framework V0.1

3.7.1. Input pins:

3.7.1.1. signal_in pin 0 :
Input to this pin is from output pin 0 of the MUX object.

3.7.2. Output pins:

3.7.2.1. signal_out pin 0 :
This pin goes to input pin 0 of the BER object.

3.8. BER (set of one to m objects):

This object is required to gather and package information from the 1 to n handset objects and the MUD. The inputs are placed into packets(???) or into arrays (???) To Be Determined (TBD). This object should be morphed into the best approximation of the packaging to be required by a targeted modem. It also compares the raw input data and raw received data. It also does the FEC detection and correction and Block error rate.

Each object in the group has one to n input pins and one to n+1 output pins. They collectively perform the:

1. Package rake/MUD information into simulated modem destination data.
2. Perform all of the bit level processing, interleaving, FEC, -- This should be in a separate block.
3. BER, BLER etc. BLER should be done via the CRC check, after all symbol decoding is performed.
~~3.~~4. Outputs to GUI input 0 pin to display the stats.
~~4.~~5. Outputs the generate the next slot command to the one to n handsets.

3.8.1. Input pins:

3.8.1.1. signal_in pin 0 to m:
Input to this pin is from output pin 0 {for now until MUD integrated} of the MUD object, or another output pin 0 of a BER object.

3.8.2. Output pins:

3.8.2.1. stats_out pin 0 :
This pin goes to input pin 0 of the host object for display of data on the GUI.

3.8.2.2. next_slot pin 1 (one of n):
This pin goes to input pin 1 of the handset object to indicate the system is ready for the next slot of data.

From: Jon Greene <greene@mc.com>
To: "Lauginiger, Frank" <fpl@mc.com>, <joates@mc.com>, <afuchs@mc.com>, <mvinskus@mc.com>
Date: 6/23/00 3:05PM
Subject: Some MUD analysis

All:

Obviously, I've been thinking about MUD a lot. Below is some analysis.

First, some news. We apparently have 400 Mhz, 2 meg / 266 Mhz L2 Nitros in house (samples). Vitaly is presently working to bring them up. This is excellent news. Besides the above speed/size properties, Nitros use significantly lower power than Max's and allow for varying L2 configuration options. Nitro L2's can be configured the normal way (as a cache) or all or half (1 meg) as SRAM memory and can be addressed as such directly. For example, one can write a buffer into this memory with vmov or, better yet, as the output of some computation. I'm not sure if it could be the source or target of a RACEway xfer but we should try to find this out. Even if configured as a coherent cache, it can be easily locked and unlocked in user mode. I think configuring as 2 meg of SRAM may work the best for MUD but we should determine this empirically.

Now, a critical analysis of ops, buffer sizes, bandwidth, access patterns, algorithm structure and phases of the moon, are all essential to arriving at a strategy that stands a chance of working. This of course is not easy because various techniques impact all of the above in unequal ways. Let's just consider the R1/R1m R-matrix processing on the above Nitro with a maximum of 100 users. *Without* taking advantage of the diagonal symmetry in the Corr matrix, which I now believe will be very difficult to do in the R-matrix ucoded processing loop(s) (we should discuss this), but still assuming Corr *can* effectively exist as a byte matrix without degrading accuracy beyond acceptability, a single plane (i.e., a processor's worth) of the Corr matrix requires 200 * 200 * 32 = 1,280,000 bytes which fits, albeit uncomfortably, into the L2. At 2 gigabyes/sec (~ 266 * 8), this matrix (if L2 resident) can theoretically be consumed in 0.64 ms (remember, 1.33 ms. is our budget). Now, *if* we go with a completely separate X matrix calculation without stripmining *and* we also store it as byte values, it would require at most 100 * 100 * 32 = 320,000 bytes. This must be entirely produced and consumed in the 1.33 ms. time slice. In *theory*, this can be done in 0.32 ms. Finally, the R1_temp output is of size 200 * 200 = 40,000 bytes and can be produced in .02 ms. So, with the fully separate X matrix approach and no symmetry in the Corr, we theoretically require ~1,750,000 bytes of buffer size (I added a little more for stray stuff such as the C vectors and the phys <=> virt Luts, etc.) and ~1.0 ms. to produce and consume these buffers. If we stripmined X, which seems a better way to go, we could hopefully keep it resident in L1, thereby reducing L2 buffers to ~1,350,000 bytes and 0.7 ms of L2 I/O. The stripmining also allows us the option of keeping the X strip as shorts rather than bytes.

Now lets consider the ops count. For the R1/R1m processing (including the generation of the X matrix and 2 antennas), I come up with (2 * 6 * 100 * 100 * 16 + 4 * 200 * 200 * 16) * 750 = (1920,000 + 2,560,000) * 750 = 3.36 GOPS. (BTW, if you were wondering, 750 = 1000/1.33.) The R0 processing has less GOPS due to the symmetry. I get (1920,000 + 2,560,000/2) * 750 = 2.40 GOPS. Since the R0 and R1/R1m processing use the same X matrix, we may be

tempted to consider having only the R0 processor compute the X matrix and ship it to the R1/R1m processor. This looks nice from a GOPS perspective (R0 = 2.40, R1/R1m = 1.92) but I'm not sure it will work very well given the lockstep nature of the processing pipe. For example, will the R1/R1m processor simply be idle waiting for the X matrix or will it be completing the *prior* R1_temp processing while the R0 processor is computing the current X?

But the real killer about having R0 ship X to R1/R1m is that the X matrix (320,000 bytes) will take at least 1.23 ms. over RACE++ (320,000/260,000,000). And let's not forget the 40,000 byte R_temp output matrix that has to also be shipped out in the same time frame. So I don't think this OPs balancing approach will work.

We therefore appear to require 3.36 GOPS out of R1/R1m and we might just not even bother with the R0 symmetry since it doesn't buy you very much given that mpic needs both R0 and R1/R1m as inputs. In other words, have both R-matrix processors run essentially the same code. (Will this work?)

Now 3.36 GOPS out of one processor is a tall order. We may have to resort to a more asymmetric division of labor (The R0 processor takes advantage of the R0 symmetry and also does a portion of R1/R1m). But, I'd like to pursue the more balanced division until we are absolutely sure it won't work.

It this approach, both the R0 and R1/R1m processors independently produce and consume X in strips. A variant could instead produce and consume a single "value" (actually 32 shorts) of X in a single ucode primitive that does both the complex multiplies and the dot products (the MUDder of all primitives). The former is certainly the easier approach and might get us all the way there but the latter, if it can be cleverly coded, may perform better. In all cases, the ops don't change but at least the L2 gets some breathing room.

In any event, the so-called dot-product loop, whether it's separate or includes the complex multiply, still remains a difficult piece of code to fully optimize if we allow the number of virtual to physical users to vary as MUD (and Dr. Oates) demands. Using a LUT to acquire the index list and count of virtual users for a given physical user will tend to throttle the dot product code due to short vector lengths, funny address calculations, and "random" load and store patterns. The load isn't so bad since it's two cache lines no matter where it comes from. We may want to reorder Corr anyway just to ease the address arithmetic and DST logic. We could also simply store in the order we produce and leave it to the mpic processor to reorder (poor guy). As for the short vector count, I think this can be overcome with a clever primitive that "pauses" as little as possible between index lists but this will take some careful design.

I think we should try for the "balanced" stripmine approach with essentially the same two primitives running in each processor. In the absence of dissenting views, I will continue modifying the C code to realize this structure. I'm still not sure where the Amp/fac_xx multiply(s)/shift(s) belong but for now I'll rid them entirely from the R-matrix functions that I'm preparing for ucoding.

- Jon

CC: "Kenny , Jamie" <jfk@mc.com>

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: Channel Estimation Date: October 20, 2000

1. Introduction

In the conventional RAKE receiver, channel amplitude[1] estimation is required for maximal ratio combining the RAKE fingers. The BER performance is not strongly dependent on the accuracy of the channel amplitude estimates. For Multi-User Detection (MUD) the channel amplitude estimates are used for signal subtraction, and accuracy of the channel amplitude estimates is more critical. In addition, the channel estimation error is larger when MUD is used since channel estimation is performed in a higher interference environment. This report investigates the accuracy of the conventional channel amplitude estimation techniques under elevated multiple access interference. The effect of channel amplitude estimation error on MUD efficiency is then assessed. The analysis presented here is intended to be a first-look. There are a number of ways to increase the channel amplitude estimation accuracy. A few of these are discussed below.

Section 2 presents a model for the received signal and match-filter outputs. The effect of channel estimation error on MUD efficiency is addressed in section 3. In section 4 the accuracy of the conventional channel amplitude estimates is assessed. In section 5 improved single-user methods are presented for channel amplitude estimation. Section 6 presents a multi-user channel amplitude estimation method. Section 7 addresses the effect of uncancelled multipath on the MUD efficiency, which is used in section 8 to assess the effect of dropping small amplitudes. It is shown that the overall MUD efficiency is improved by dropping small amplitudes. Conclusions are drawn in section 9.

2. Signal Model and Matched-Filter Outputs

The baseband received signal can be written

[1] Amplitudes are complex and hence include magnitude and phase.

$$r[t]=\sum_{k=1}^{K_v}\sum_{m}\tilde{s}_k[t-mT]b_k[m]+w[t] \tag{1}$$

where t is the integer time sample index, $T = NN_c$ is the data bit duration, $N = 256$ is the short-code length, N_c is the number of samples per chip, $w[t]$ is receiver noise, and where $\tilde{s}_k[t]$ is the channel-corrupted signature waveform for virtual user k. For L multipath components the channel-corrupted signature waveform for virtual user k is modeled as

$$\tilde{s}_k[t]=\sum_{p=1}^{L}a_{kp}s_k[t-\tau_{kp}] \tag{2}$$

where a_{kp} are the complex multipath amplitudes. Notice that $a_{kp} = a_{lp}$ if k and l are two virtual users corresponding to the same physical user. This is due to the fact that the signal waveforms of all virtual users corresponding to the same physical user pass through the same channel. For multiple antennas a_{kp} is a vector. For dual antennas, for example, primary and diversity,

$$a_{kp}=\begin{bmatrix}a_{p,kp}\\ a_{d,kq}\end{bmatrix} \tag{3}$$

The waveform $s_k[t]$ is referred to as the signature waveform for the kth virtual user. This waveform is generated by passing the spreading code sequence $c_k[n]$ through a pulse-shaping filter $g[t]$

$$s_k[t]=\sum_{r=0}^{N-1}g[t-rN_c]c_k[r] \tag{4}$$

where $N = 256$ and $g[t]$ is the raised-cosine pulse shape. Since $g[t]$ is a raised-cosine pulse as opposed to a root-raised-cosine pulse, the received signal $r[t]$ represents the baseband signal after filtering by the matched chip filter. Note that for spreading factors less than 256 some of the chips $c_k[r]$ are zero.

Combining Equations (1) through (4) gives

$$r[t]=\sum_{k=1}^{K_v}\sum_{\overline{m}}\sum_{p=1}^{L}a_{kp}s_k[t-\overline{m}T-\tau_{kp}]b_k[\overline{m}]+w[t] \tag{5}$$

The output of the despreading operation for a single multipath component is the complex statistic

$$
\begin{aligned}
y_{lq}[m] &\equiv \frac{1}{2N_l}\sum_n r[nN_c+\hat{\tau}_{lq}+mT]\cdot c_l^*[n] \\
&= \sum_{k=1}^{K_v}\sum_{\overline{m}}\sum_{p=1}^{L} a_{kp}\left\{\frac{1}{2N_l}\sum_n s_k[nN_c+(m-\overline{m})T+\hat{\tau}_{lq}-\tau_{kp}]\cdot c_l^*[n]\right\}\cdot b_k[\overline{m}]+w_{lq}[m] \\
&= \sum_{k=1}^{K_v}\sum_{m'}\sum_{p=1}^{L} a_{kp}\cdot C_{lkqp}[m']\cdot b_k[m-m']+w_{lq}[m]
\end{aligned} \tag{6}
$$

$$
\begin{aligned}
C_{lkqp}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c+m'T+\hat{\tau}_{lq}-\tau_{kp}]\cdot c_l^*[n] \\
w_{lq}[m] &\equiv \frac{1}{2N_l}\sum_n w[nN_c+\hat{\tau}_{lq}+mT]\cdot c_l^*[n]
\end{aligned}
$$

where $\hat{\tau}_{lq}$ is the estimate of τ_{lq}, and N_l is the (non-zero) length of code $c_l[n]$. The values $y_{lq}[m]$ are complex and are referred to as the pre-MRC matched-filter outputs. For multiple antennas, $r[t]$, $w[t]$, $y_{lq}[m]$ and $w_{lq}[m]$ are column vectors.

The matched-filter output is then

$$
\begin{aligned}
y_l[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^H\cdot y_{lq}[m]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^H\cdot\sum_{k=1}^{K_v}\sum_{m'}\sum_{p=1}^{L} a_{kp}\cdot C_{lkqp}[m']\cdot b_k[m-m']+\sum_{q=1}^{L}\hat{a}_{lq}^H\cdot w_{lq}[m]\right\} \\
&= \sum_{k=1}^{K_v}\sum_{m'}\mathrm{Re}\left\{\sum_{q=1}^{L}\sum_{p=1}^{L}\hat{a}_{lq}^H a_{kp}\cdot C_{lkqp}[m']\right\}\cdot b_k[m-m']+w_l[m] \\
&= \sum_{k=1}^{K_v}\sum_{m'} r_{lk}[m']\cdot b_k[m-m']+w_l[m]
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
r_{lk}[m'] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\sum_{p=1}^{L}\hat{a}_{lq}^H a_{kp}\cdot C_{lkqp}[m']\right\} \\
w_l[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^H\cdot w_{lq}[m]\right\}
\end{aligned}
$$

where $\hat{a}_{lq}^H$ is the estimate of a_{lq}^H and $w_l[m]$ is the match-filtered receiver noise. The terms for $m'\neq 0$ result from asynchronous users.

3. Effect of Amplitude Estimation Error on MUD Efficiency

MUD efficiency is defined in terms of the ratio of the intra-cell interference with MUD (I_{MUD}) to the intra-cell interference with the Matched Filter (MF), that is, the intra-cell interference without MUD (I_{MF}):

$$\beta_{MUD} \equiv 1 - \frac{I_{MUD}}{I_{MF}} \tag{8}$$

The total interference without MUD is $I_{MF} + J$, where J is the inter-cell interference. Similarly, the total interference with MUD is $I_{MUD} + J$. The ratio of inter-cell interference to intra-cell interference without MUD is denoted $f = J/I_{MF}$. The increase in system capacity is equal to the ratio of the total interference without MUD to the total interference with MUD, which is $(I_{MF} + J)/(I_{MUD} + J) = (I_{MF} + fI_{MF})/(I_{MUD} + fI_{MF}) = (1 + f)/(1 - \beta_{MUD} + f)$. For $f = 0.3$ and $\beta_{MUD} = 0.7$, MUD increases the system capacity by a factor of 1.3/(1 – 0.7 +0.3) = 2.2. Hence, if our goal is to double system capacity the MUD efficiency must be approximately 70% or greater.

In the following we estimate the loss in MUD efficiency, $1 - \beta_{MUD}$, due to imperfect channel estimation. For simplicity of presentation we consider approximately synchronous users.

Recall that in a synchronous system the matched-filter outputs can be expressed as

$$y_l = r_{ll} b_l + \sum_{k=1, k \neq l}^{K_v} r_{lk} b_k + \eta_l \tag{9}$$

and that the intra-cell interference is then

$$I_{MF} = \sum_{k=1, k \neq l}^{K_v} E\{r_{lk}^2\} \tag{10}$$

The effect of channel amplitude errors is that the estimates of the R-matrix elements (r_{lk}) are imperfect, which reduces the interference that is cancelled. When MUD is employed with imperfect R-matrix estimates the detection statistic is

$$\begin{aligned} y_l - \sum_{k=1,k\neq l}^{K_v} \hat{r}_{lk} \hat{b}_k &= \\ &= A_l^2 b_l + \sum_{k=1,k\neq l}^{K_v} r_{lk} b_k - \sum_{k=1,k\neq l}^{K_v} \hat{r}_{lk} \hat{b}_k + \eta_l \\ &= A_l^2 b_l + \sum_{k=1,k\neq l}^{K_v} (r_{lk} - \hat{r}_{lk}) b_k + \eta_l \end{aligned} \tag{11}$$

where for the present case we have assumed that the bit estimates are perfect. With MUD the intra-cell interference is

$$I_{MUD} \equiv \sum_{k=1,k\neq l}^{K_v} \sum_{k'=1,k'\neq l}^{K_v} E\{(r_{lk} - \hat{r}_{lk})(r_{lk'} - \hat{r}_{lk'})\} E\{b_k b_{k'}\}$$
$$= \sum_{k=1,k\neq l}^{K_v} E\{(r_{lk} - \hat{r}_{lk})^2\} \tag{12}$$

Now from Equation (7), specialized for synchronous users

$$r_{lk} = \mathrm{Re}\left\{\sum_{q=1}^{L}\sum_{p=1}^{L} \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp}\right\}$$
$$= \frac{1}{2}\sum_{q=1}^{L}\sum_{p=1}^{L}\left\{\hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^*\right\}$$
$$\hat{r}_{lk} = \frac{1}{2}\sum_{q=1}^{L}\sum_{p=1}^{L}\left\{\hat{a}_{lq}^H \hat{a}_{kp} \cdot C_{lkqp} + \hat{a}_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^*\right\}$$
$$r_{lk} - \hat{r}_{lk} = \frac{1}{2}\sum_{q=1}^{L}\sum_{p=1}^{L}\left\{\hat{a}_{lq}^H\left[a_{kp} - \hat{a}_{kp}\right] \cdot C_{lkqp} + \left[a_{kp}^H - \hat{a}_{kp}^H\right]\hat{a}_{lq} \cdot C_{lkqp}^*\right\} \tag{13}$$
$$= \frac{1}{2}\sum_{q=1}^{L}\sum_{p=1}^{L}\left\{\hat{a}_{lq}^H \varepsilon_{kp} \cdot C_{lkqp} + \varepsilon_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^*\right\}$$

$$\varepsilon_{kp} \equiv a_{kp} - \hat{a}_{kp}$$

Hence the second-order statistics are

$$E\{(r_{lk} - \hat{r}_{lk})^2\} = \frac{1}{4}\sum_{q,p=1}^{L}\sum_{q',p'=1}^{L} E\left\{\left[\hat{a}_{lq}^H \varepsilon_{kp} C_{lkqp} + \varepsilon_{kp}^H \hat{a}_{lq} C_{lkqp}^*\right] \cdot \left[\hat{a}_{lq'}^H \varepsilon_{kp'} C_{lkq'p'} + \varepsilon_{kp'}^H \hat{a}_{lq'} C_{lkq'p'}^*\right]\right\}$$
$$= \frac{1}{4}\sum_{q,p=1}^{L}\sum_{q',p'=1}^{L} E\left\{\hat{a}_{lq}^H \varepsilon_{kp} C_{lkqp} \cdot \varepsilon_{kp'}^H \hat{a}_{lq'} C_{lkq'p'}^* + \varepsilon_{kp}^H \hat{a}_{lq} C_{lkqp}^* \cdot \hat{a}_{lq'}^H \varepsilon_{kp'} C_{lkq'p'}\right\}$$
$$= \frac{1}{4N_l}\sum_{q,p=1}^{L} E\left\{\hat{a}_{lq}^H \varepsilon_{kp} \cdot \varepsilon_{kp}^H \hat{a}_{lq} + \varepsilon_{kp}^H \hat{a}_{lq} \cdot \hat{a}_{lq}^H \varepsilon_{kp}\right\}$$
$$\cong \frac{1}{2N_l}\sum_{q,p=1}^{L} E\left\{a_{lq}^H \varepsilon_{kp} \cdot \varepsilon_{kp}^H a_{lq}\right\} = \frac{1}{2N_l}\sum_{q,p=1}^{L} Tr\left[E\left\{a_{lq} a_{lq}^H\right\} \cdot E\left\{\varepsilon_{kp}\varepsilon_{kp}^H\right\}\right]$$
$$= \frac{1}{2N_l}\sum_{q=1}^{L} A_{lq}^2 \cdot \sum_{p=1}^{L} \mathrm{E}_{kp}^2 \cdot \left[2 + 2\mathrm{Re}(\rho_{lq}\rho_\varepsilon^*)\right]$$
$$\cong \frac{1}{2N_l} A_l^2 \cdot \mathrm{E}_k^2 \cdot 2 \cdot \left[1 + |\rho|^2\right]$$

$$A_{lq}^2 \equiv E\left\{|a_{p,lq}|^2\right\} \equiv E\left\{|a_{d,lq}|^2\right\} \quad \mathrm{E}_{kp}^2 \equiv E\left\{|\varepsilon_{p,kp}|^2\right\} \equiv E\left\{|\varepsilon_{d,kp}|^2\right\}$$
$$A_l^2 \equiv \sum_{q=1}^{L} A_{lq}^2, \quad \mathrm{E}_k^2 \equiv \sum_{p=1}^{L} \mathrm{E}_{kp}^2 \tag{14}$$
$$\rho \approx \rho_{lq} \approx \rho_\varepsilon^*$$

where we have assumed that the amplitude error is independent of the amplitude and we have used

$$E\{a_{lq}\cdot a_{lq}^H\}=E\left\{\begin{bmatrix}a_{p,lq}\\ a_{d,lq}\end{bmatrix}\cdot\begin{bmatrix}a_{p,lq}^* & a_{d,lq}^*\end{bmatrix}\right\}=A_{lq}^2\cdot\begin{bmatrix}1 & \rho_{lq}\\ \rho_{lq}^* & 1\end{bmatrix}$$
$$E\{\varepsilon_{kp}\cdot\varepsilon_{kp}^H\}=E\left\{\begin{bmatrix}\varepsilon_{p,kp}\\ \varepsilon_{d,kp}\end{bmatrix}\cdot\begin{bmatrix}\varepsilon_{p,kp}^* & \varepsilon_{d,kp}^*\end{bmatrix}\right\}=\mathrm{E}_{kp}^2\cdot\begin{bmatrix}1 & \rho_\varepsilon\\ \rho_\varepsilon^* & 1\end{bmatrix} \qquad (15)$$

The second expression is discussed below. We refer to E_k as the error amplitude for the kth virtual user. The residual interference after MUD IC is

$$\begin{aligned}I_{MUD} &= \sum_{k=1,k\neq l}^{K_v} E\{(r_{lk}-\hat{r}_{lk})^2\}\\ &=\frac{A^2}{2N_l}\left[(K-1)\alpha\mathrm{E}_d^2+K\mathrm{E}_c^2\right]\cdot 2\cdot\left[1+|\rho|^2\right]\\ &\cong\frac{A^2K}{2N_l}\left[\alpha+\beta_c^2\right]\mathrm{E}_d^2\cdot 2\cdot\left[1+|\rho|^2\right]\end{aligned} \qquad (16)$$

where all data channels have amplitude A. The error amplitude for the control channels is denoted E_c and the error amplitude for the data channels is denoted E_d. All data channel amplitudes are determined by scaling the corresponding control channel amplitudes by $1/\beta_c$. Hence $\mathrm{E}_d=\mathrm{E}_c/\beta_c$.

Similarly we can show that

$$E\{r_{lk}^2\}=\frac{1}{2N_l}A_l^2\cdot A_k^2\cdot 2\cdot\left[1+|\rho|^2\right] \qquad (17)$$

so that the matched-filter interference is

$$\begin{aligned}I_{MF} &= \sum_{k=1,k\neq l}^{K_v} E\{r_{lk}^2\}\\ &=\frac{A^2}{2N_l}\left[(K-1)\alpha A^2+K\beta_c^2A^2\right]\cdot 2\cdot\left[1+|\rho|^2\right]\\ &\cong\frac{A^2K}{2N_l}\left[\alpha+\beta_c^2\right]A^2\cdot 2\cdot\left[1+|\rho|^2\right]\end{aligned} \qquad (18)$$

Finally, the MUD efficiency is

$$\beta_{MUD}\equiv 1-\frac{I_{MUD}}{I_{MF}}=1-\left(\frac{\mathrm{E}_d}{A}\right)^2 \qquad (19)$$

4. Conventional Channel Estimation

The conventional channel amplitude estimate is given by

$$\begin{aligned}\hat{a}_{lq} &= \frac{1}{M}\sum_{m=1}^{M} y_{lq}[m]\cdot b_l[m] \\ &= \sum_{k=1}^{K_v}\sum_{p=1}^{L} a_{kp}\cdot\sum_{m'} C_{lkqp}[m']\frac{1}{M}\sum_{m=1}^{M} b_l[m]\cdot b_k[m-m'] + \frac{1}{M}\sum_{m=1}^{M} w_{lq}[m]\cdot b_l[m] \\ &= \sum_{k=1}^{K_v}\sum_{p=1}^{L} H_{lqkp}\cdot a_{kp} + w_{lq}\end{aligned} \tag{20}$$

where

$$\begin{aligned}H_{lqkp} &\equiv \sum_{m'} C_{lkqp}[m']\cdot I_{lk}[m'] \\ I_{lk}[m'] &\equiv \frac{1}{M}\sum_{m=1}^{M} b_l[m]\cdot b_k[m-m'] \\ w_{lq} &\equiv \frac{1}{M}\sum_{m=1}^{M} w_{lq}[m]\cdot b_l[m]\end{aligned} \tag{21}$$

In the above $b_l[m]$ represent the known pilot bits. (The lth virtual user is implicitly a control channel.) The number M represents the number of pilot bits used to derive the channel amplitude estimates. The channel amplitude estimate can be rewritten

$$\begin{aligned}\hat{a}_{lq} &= \sum_{k=1}^{K_v}\sum_{p=1}^{L} H_{lqkp}\cdot a_{kp} + w_{lq} \\ &= \sum_{p=1}^{L} H_{lqlp}\cdot a_{lp} + \sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L} H_{lqkp}\cdot a_{kp} + w_{lq} \\ &= a_{lq} + \sum_{\substack{p=1\\p\neq q}}^{L} H_{lqlp}\cdot a_{lp} + \sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L} H_{lqkp}\cdot a_{kp} + w_{lq}\end{aligned} \tag{22}$$

It is shown in the appendix that

$$E\{H_{lqkp}\cdot H^{*}_{l'q'k'p'}\}_{lq\neq kp} = \delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}E\{|H_{lqkp}|^2\}_{lq\neq kp} \tag{23}$$

Hence the variance of the estimate is

$$E\{\varepsilon_{x,lq}\cdot\varepsilon_{y,lq}^{*}\}=\sum_{\substack{p=1\\p\neq q}}^{L}E\{|H_{lqlp}|^{2}\}\cdot E\{a_{x,lp}\cdot a_{y,lp}^{*}\}$$
$$+\sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L}E\{|H_{lqkp}|^{2}\}\cdot E\{a_{x,kp}\cdot a_{y,kp}^{*}\}+E\{w_{x,lq}\cdot w_{y,lq}^{*}\}$$

$$E\{|\varepsilon_{p,lq}|^{2}\}=E\{|\varepsilon_{d,lq}|^{2}\}=\sum_{\substack{p=1\\p\neq q}}^{L}E\{|H_{lqlp}|^{2}\}\cdot A_{lp}^{2}+\sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L}E\{|H_{lqkp}|^{2}\}\cdot A_{kp}^{2}+W_{lq}^{2}\equiv \mathrm{E}_{lq}^{2} \tag{24}$$
$$E\{\varepsilon_{p,lq}\cdot\varepsilon_{d,lq}^{*}\}=\sum_{\substack{p=1\\p\neq q}}^{L}E\{|H_{lqlp}|^{2}\}\cdot \rho_{lp}A_{lp}^{2}+\sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L}E\{|H_{lqkp}|^{2}\}\cdot \rho_{kp}A_{kp}^{2}\approx \rho_{\varepsilon}\mathrm{E}_{lq}^{2}$$

$$W_{lq}^{2}\equiv E\{|w_{p,lq}|^{2}\}=E\{|w_{d,lq}|^{2}\}$$

The factor ρ_ε simply reflects the fact that the off-diagonal elements are smaller than the diagonal elements due to partial correlations ρ_{kp} between the antenna elements. In the Appendix it is also shown that

$$\begin{aligned}E\{|H_{lqlp}|^{2}\}_{q\neq p}&\cong\frac{1}{N_l}\\ E\{|H_{lqkp}|^{2}\}_{l\neq k}&=\frac{1}{MN_l}\end{aligned} \tag{25}$$

Now combining Equations (24) and (25) gives for the variance of the channel amplitude estimate

$$\begin{aligned}\mathrm{E}_{lq}^{2}&=\sum_{\substack{p=1\\p\neq q}}^{L}E\{|H_{lqlp}|^{2}\}\cdot A_{lp}^{2}+\sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L}E\{|H_{lqkp}|^{2}\}\cdot A_{kp}^{2}+W_{lq}^{2}\\ &=\frac{1}{N_l}\sum_{\substack{p=1\\p\neq q}}^{L}A_{lp}^{2}+\frac{1}{MN_l}\sum_{\substack{k=1\\k\neq l}}^{K_v}\sum_{p=1}^{L}A_{kp}^{2}+W_{lq}^{2}\\ &=\frac{1}{N_l}\cdot\frac{L-1}{L}A_l^{2}+\frac{1}{MN_l}\sum_{\substack{k=1\\k\neq l}}^{K_v}A_k^{2}+W_{lq}^{2}\end{aligned} \tag{26}$$

where we have used $A_{lp}^{2} = A_l^{2}/L$. The first term represents the variance due to a user's own multipath interference. This term is small compared to the variance arising from the total multiple-access interference. For simplicity we incorporate part of this term into the second term and drop the remainder. The final term represents thermal noise and other-cell interference. For now we assume that thermal noise in small. The interference arising from other cells is assumed to be proportional to the same-cell interference, with a constant of proportionality $f = 0.35$. With these assumptions we have

$$E_l^2 = \sum_{q=1}^{L} E_{lq}^2 = (1+f)\frac{L}{MN_l}\sum_{k=1}^{K_v} A_k^2 \tag{27}$$

Notice that the magnitude of the error E_l is approximately the same for all users. Also, the lth users is implicitly a control channel, and hence $N_l = PG = 256$. If the K_v virtual users are all at the highest spreading factor, then in terms of the $K = K_v/2$ physical users we have

$$E_c^2 = (1+f)\frac{L}{M \cdot PG}\left[K\beta_c^2 A^2 + K\alpha A^2\right] \tag{28}$$

where E_c is the magnitude of the channel amplitude error for a control channel, β_c is the relative control channel amplitude, A is the amplitude for the data channels, and where α is the activity factor for the data channels. Since the channel amplitudes for the data channels are determined by scaling the amplitude of the corresponding control channel it is evident that $E_d = E_c/\beta_c$. Hence,

$$\left(\frac{E_d}{A}\right)^2 = (1+f)\frac{KL}{M \cdot PG}\left[1+\frac{\alpha}{\beta_c^2}\right] \tag{29}$$

Given the parameters

f	$= 0.35$
K	$= 128$
L	$= 4$
M	$= 18$
PG	$= 256$
α	$= 0.4$
β_c	$= 0.7333$

we get

$$\begin{aligned}\frac{E_d}{A} &= \sqrt{(1+f)\frac{KL}{M \cdot PG}\left[1+\frac{\alpha}{\beta_c^2}\right]} \\ &= \sqrt{(1+0.35)\frac{(128)(4)}{(18)(256)}\left[1+\frac{0.40}{(0.7333)^2}\right]} \\ &= 0.51\end{aligned} \tag{30}$$

The number of pilot bits, M, is taken to be 18, which represents 6 bits per slot, the amplitudes averaged over 3 slots. The corresponding MUD efficiency is

$$\beta_{MUD} = 1-\left(\frac{E_d}{A}\right)^2 = 1-(0.51)^2 = 0.74 \tag{31}$$

5. Improved Channel Amplitude Estimates

One method for significantly improving the channel amplitude estimates is to perform a second estimate directly on the data channels after the initial data channel demodulation. Performance is improved for two reasons. First, the entire slot can be used for integration. Hence we have $M = 3(10) = 30$ bits. Secondly, the error is not scaled by $1/\beta_c$ since the estimate is performed directly on the data channel. For this method we have

$$\begin{aligned}\frac{E_d}{A} &= \sqrt{(1+f)\frac{KL}{M \cdot PG}\left[\beta_c^2 + \alpha\right]} \\ &= \sqrt{(1+0.35)\frac{(128)(4)}{(30)(256)}\left[(0.7333)^2 + 0.40\right]} \\ &= 0.29\end{aligned} \tag{32}$$

and the corresponding MUD efficiency is

$$\beta_{MUD} = 1 - \left(\frac{E_d}{A}\right)^2 = 1 - (0.29)^2 = 0.92 \tag{33}$$

Slightly better performance can be achieved by using both data and control channels. This method can be performed either on the daughter card or on the modem card since it is a single user method. The assumption is that the matched-filter BER is sufficiently good.

6. Multiuser Channel Amplitude Estimation

Given the conventional channel estimates and the detected user bits it is possible to subtract the MAI which corrupts channel estimation. This method of channel estimation is referred to as multiuser channel estimation, as opposed to the conventional single-user estimation techniques. A simple multiuser channel estimation technique is presented below without analysis. Performance should be determined via simulation.

From Equation (22) the conventional estimate is

$$\hat{a}_{lq} = \sum_{kp} H_{lqkp} \cdot a_{kp} + w_{lq} \tag{34}$$

A multiuser estimate is obtained by subtracting the known interference among the channel estimates

$$
\begin{aligned}
\hat{a}_{lq} &= a_{lq} + \sum_{kp \neq lq} H_{lqkp} \cdot a_{kp} + w_{lq} \\
\hat{\hat{a}}_{lq} &\equiv \hat{a}_{lq} - \sum_{k'p' \neq lq} H_{lqk'p'} \cdot \hat{a}_{k'p'} \\
&= \left[a_{lq} + \sum_{kp \neq lq} H_{lqkp} \cdot a_{kp} + w_{lq} \right] - \sum_{k'p' \neq lq} H_{lqk'p'} \left[a_{k'p'} + \sum_{kp \neq k'p'} H_{k'p'kp} \cdot a_{kp} + w_{k'p'} \right] \\
&= a_{lq} - \left[\sum_{k'p' \neq lq} \sum_{kp \neq k'p'} H_{lqk'p'} \cdot H_{k'p'kp} \cdot a_{kp} \right] + \left[w_{lq} - \sum_{k'p' \neq lq} H_{lqk'p'} w_{k'p'} \right]
\end{aligned}
\tag{35}
$$

where the (hopefully) improved multiuser channel estimate is denoted $\hat{\hat{a}}_{lq}$. The first term above is the actual channel amplitude. The second term is the residual interference, and the last term represents thermal noise and other-cell interference, which is amplified by the multiuser interference subtraction. The extent of the amplification needs to be determined.

7. Effect of Uncancelled Multipath Interference

It is expected that a typical RAKE receiver will be capable of tracking up to approximately 16 multipath components. Since the computational complexity of symbol-rate MUD is quadratic in the number of multipaths L it is unlikely that MUD implementations will be able to cancel all multipath interference. The effect of uncancelled multipath is assessed below.

Suppose that the RAKE receiver processes L' multipath components, but that the MUD implementation cancels interference for $L < L'$ components. From Equation (13) we have

$$
\begin{aligned}
r_{lk} &= \frac{1}{2} \sum_{q=1}^{L'} \sum_{p=1}^{L'} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} \\
&= \frac{1}{2} \sum_{q=1}^{L} \sum_{p=1}^{L} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} + \frac{1}{2} \sum_{q=1}^{L} \sum_{p=L+1}^{L'} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} \\
&+ \frac{1}{2} \sum_{q=L+1}^{L'} \sum_{p=1}^{L} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} + \frac{1}{2} \sum_{q=L+1}^{L'} \sum_{p=L+1}^{L'} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} \\
\hat{r}_{lk} &= \frac{1}{2} \sum_{q=1}^{L} \sum_{p=1}^{L} \left\{ \hat{a}_{lq}^H \hat{a}_{kp} \cdot C_{lkqp} + \hat{a}_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\}
\end{aligned}
\tag{36}
$$

$$
\begin{aligned}
r_{lk} - \hat{r}_{lk} &= \frac{1}{2} \sum_{q=1}^{L} \sum_{p=1}^{L} \left\{ \hat{a}_{lq}^H \varepsilon_{kp} \cdot C_{lkqp} + \varepsilon_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} + \frac{1}{2} \sum_{q=1}^{L} \sum_{p=L+1}^{L'} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} \\
&+ \frac{1}{2} \sum_{q=L+1}^{L'} \sum_{p=1}^{L} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\} + \frac{1}{2} \sum_{q=L+1}^{L'} \sum_{p=L+1}^{L'} \left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{lkqp} + a_{kp}^H \hat{a}_{lq} \cdot C_{lkqp}^* \right\}
\end{aligned}
$$

and the variance is then

$$
\begin{aligned}
E\{(r_{lk}-\hat{r}_{lk})^2\} &= \frac{1}{2N_l}\sum_{q=1}^{L}\sum_{p=1}^{L}\{\hat{a}_{lq}^H\varepsilon_{kp}\varepsilon_{kp}^H\hat{a}_{lq}\}+\frac{1}{2N_l}\sum_{q=1}^{L}\sum_{p=L+1}^{L'}\{\hat{a}_{lq}^H a_{kp}a_{kp}^H\hat{a}_{lq}\} \\
&+\frac{1}{2N_l}\sum_{q=L+1}^{L'}\sum_{p=1}^{L}\{\hat{a}_{lq}^H a_{kp}a_{kp}^H\hat{a}_{lq}\}+\frac{1}{2N_l}\sum_{q=L+1}^{L'}\sum_{p=L+1}^{L'}\{\hat{a}_{lq}^H a_{kp}a_{kp}^H\hat{a}_{lq}\} \\
&\cong \frac{1}{2N_l}\sum_{q=1}^{L}\sum_{p=1}^{L}\{a_{lq}^H\varepsilon_{kp}\varepsilon_{kp}^H a_{lq}\}+\frac{1}{2N_l}\sum_{q=1}^{L}\sum_{p=L+1}^{L'}\{a_{lq}^H a_{kp}a_{kp}^H a_{lq}\} \\
&+\frac{1}{2N_l}\sum_{q=L+1}^{L'}\sum_{p=1}^{L}\{a_{lq}^H a_{kp}a_{kp}^H a_{lq}\}+\frac{1}{2N_l}\sum_{q=L+1}^{L'}\sum_{p=L+1}^{L'}\{a_{lq}^H a_{kp}a_{kp}^H a_{lq}\} \\
&\cong \frac{2\cdot[1+|\rho|^2]}{2N_l}\left\{\sum_{q=1}^{L}\sum_{p=1}^{L}A_{lq}^2\mathrm{E}_{kp}^2+\sum_{q=1}^{L}\sum_{p=L+1}^{L'}A_{lq}^2A_{kp}^2+\sum_{q=L+1}^{L'}\sum_{p=1}^{L}A_{lq}^2A_{kp}^2+\sum_{q=L+1}^{L'}\sum_{p=L+1}^{L'}A_{lq}^2A_{kp}^2\right\} \\
&= \frac{2\cdot[1+|\rho|^2]}{2N_l}\left\{\sum_{q=1}^{L}A_{lq}^2\sum_{p=1}^{L}\mathrm{E}_{kp}^2+[\beta_{x,k}+\beta_{x,l}+\beta_{x,l}\beta_{x,k}]\sum_{q=1}^{L}A_{lq}^2\sum_{p=L}^{L}A_{kp}^2\right\} \\
&= \frac{2\cdot[1+|\rho|^2]}{2N_l}\{A_l^2\mathrm{E}_k^2+[\beta_{x,k}+\beta_{x,l}+\beta_{x,l}\beta_{x,k}]A_l^2A_k^2\}
\end{aligned}
$$

$$\beta_{x,k}\equiv\sum_{p=L+1}^{L'}A_{kp}^2\Big/\sum_{p=1}^{L}A_{kp}^2 \tag{37}$$

Note that $\beta_{x,k}$ is the ratio of the uncancelled to cancelled interference for the kth users. Similarly, we have

$$E\{r_{lk}^2\}=\frac{2\cdot[1+|\rho|^2]}{2N_l}\{A_l^2A_k^2+[\beta_{x,k}+\beta_{x,l}+\beta_{x,l}\beta_{x,k}]A_l^2A_k^2\} \tag{38}$$

Now, neglecting the second order terms $\beta_{x,l}\beta_{x,k}$ and averaging over the users $\beta_x = E\{\beta_{x,l}\}$ we arrive at

$$\begin{aligned} I_{MUD} &= \sum_{k=1,k\neq l}^{K_v} E\left\{(r_{lk} - \hat{r}_{lk})^2\right\} \\ &= \frac{2\cdot\left[1+|\rho|^2\right]}{2N_l} KA^2\left\{(\alpha E_d^2 + E_c^2) + 2\beta_x(\alpha A_d^2 + A_c^2)\right\} \\ &= \frac{2\cdot\left[1+|\rho|^2\right]}{2N_l} KA^2\left\{(\alpha+\beta_c^2)E_d^2 + 2\beta_x(\alpha+\beta_c^2)A^2\right\} \\ &= \frac{2\cdot\left[1+|\rho|^2\right]}{2N_l} KA^2(\alpha+\beta_c^2)\left\{E_d^2 + 2\beta_x A^2\right\} \end{aligned}$$

$$\begin{aligned} I_{MF} &= \sum_{k=1,k\neq l}^{K_v} E\left\{r_{lk}^2\right\} \\ &= \frac{2\cdot\left[1+|\rho|^2\right]}{2N_l} KA^2(\alpha+\beta_c^2)\left\{A^2 + 2\beta_x A^2\right\} \end{aligned}$$

$$\beta_{MUD} = 1 - \frac{I_{MUD}}{I_{MF}} = 1 - \frac{E_d^2 + 2\beta_x A^2}{A^2 + 2\beta_x A^2} = 1 - \frac{1}{1+2\beta_x}\left[\left(\frac{E_d}{A}\right)^2 + 2\beta_x\right] \quad (39)$$

Note that β_x is the ratio of the uncancelled to cancelled interference.

In order to assess typical value for β_x multipath models [1][2][3] were used to generate random profiles. The models are based on data collected in four areas (A, B, C, and D) in the San Francisco-Oakland bay area. Table 1 below summarizes the key results. The table shows the β_x versus the number of multipath components L.

Table 1. Ratio (β_x) of the uncancelled to cancelled interference.

β_x	$L = 8$	$L = 6$	$L = 4$	$L = 3$	$L = 2$	$L = 1$
Area A	0.0019	0.0064	0.0481	0.0961	0.2376	0.5819
Area B	0.0012	0.0086	0.0404	0.1115	0.1416	0.5749
Area C	0.0004	0.0054	0.0291	0.0948	0.1649	0.6603
Area D	0.0039	0.0128	0.0430	0.0629	0.1435	0.4890

Suppose $\beta_x = 0.05$ and $(E_d/A)^2 = 0.51^2 = 0.260$. Without taking uncancelled multipath into account we found $\beta_{MUD} = 0.74$. Taking uncancelled multipath into account we find

$$\begin{aligned} \beta_{MUD} &= 1 - \frac{1}{1+2\beta_x}\left[\left(\frac{E_d}{A}\right)^2 + 2\beta_x\right] \\ &= 1 - \frac{1}{1+2(0.05)}\left[(0.51)^2 + 2(0.05)\right] \\ &= 0.67 \end{aligned} \quad (40)$$

where a worst-case $\beta_x = 0.05$ is used.

8. Improved MUD Efficiency Due to Dropping Small Amplitudes

If small amplitude multipath components are not included in the cancellation the MUD efficiency is reduced slightly due to the additional uncancelled multipath interference, but it is also increased because of the absence error resulting from the inclusion of these small noisy estimates. The net effect is a substantial increase in the MUD efficiency. From Equation (30) we have

$$\begin{aligned}\left(\frac{\mathrm{E}_{d1}}{A}\right)^2 &\equiv (1+f)\frac{K}{M\cdot PG}\left[1+\frac{\alpha}{\beta_c^2}\right]\\ &= (1+0.35)\frac{(128)}{(18)(256)}\left[1+\frac{0.40}{(0.7333)^2}\right]\\ &= 0.065\end{aligned} \tag{41}$$

where E_{d1}^2 is the error due to a single multipath (i.e. $L = 1$). From Equation (37) it is evident that if a particular multipath amplitude satisfies $A_{kp}^2 < \mathrm{E}_{d1}^2$ then it is advantageous not to incorporate this amplitude into the cancellation since the error is greater than the amplitude. Table 2 shows the mean number of paths $E\{L\}$ which satisfy $A_{kp}^2 > \mathrm{E}_{d1}^2$ and the ratio β_x of the uncancelled to cancelled interference if only these mulitpaths are cancelled. The MUD efficiency is then calculated using

$$\beta_{MUD} = 1-\frac{1}{1+2\beta_x}\left[E\{L\}\cdot\left(\frac{\mathrm{E}_{d1}}{A}\right)^2 + 2\beta_x\right] \tag{42}$$

Table 2. Improved MUD efficiency (β_{MUD}) due to dropping small amplitudes.

	$E\{L\}$	β_x	β_{MUD}
Area A	2.0300	0.0714	0.7638
Area B	2.4660	0.0691	0.7482
Area C	2.2970	0.0680	0.7564
Area D	2.0690	0.0625	0.7748
Mean	2.2155	0.0678	0.7608

9. Conclusions

This report represents a first-look at channel estimation and the effect of errors on the MUD efficiency. Only the case where all users are at the highest spreading factor has been examined. The initial results indicate that if the conventional channel estimates are used the MUD efficiency drops to 74% due to estimation errors. If the effect of uncancelled multipath interference is also considered the MUD efficiency drops down to 67%. If small amplitude multipath components are not included in the cancellation the MUD efficiency is reduced slightly due to the additional uncancelled multipath interference, but it is also increased because of the absence error resulting from the inclusion of these small noisy estimates. The net effect is a substantial increase in the

MUD efficiency, which is increased to 76%. The actual MUD efficiency will, of course, be less due to other factors which degrade efficiency. If an improved single-user channel estimation is used the MUD efficiency can be increased to 92%. This improved method requires knowledge of the pre-MRC matched-filter outputs. It is perhaps possible to further increase the MUD efficiency by employing multiuser channel estimation. These techniques also require knowledge of the pre-MRC matched-filter outputs. The above referenced MUD efficiency numbers are based on 128 users processed by the basestation. If fewer users are allowed access to the system in order to increase range the MUD efficiency is unchanged shine the total interference and noise remains unchanged.

References

[1] G. L. Turin, F. D. Clapp, T. L. Johnston, S. B. Fine, D. Lavry, "A statistical model of urban multipath propagation," IEEE Trans. on Vehicular Technology, vol. VT-21, No. 1, February 1972, pp. 1 – 9.

[2] H. Suzuki, "A statistical model for urban radio propagation," IEEE Trans. on Communications, vol. COM-25, No. 7, July 1977, pp. 673 – 680.

[3] H. Hashemi, "Simulation of the urban radio propagation channel," IEEE Trans. Vehicular Technology, vol. VT-28, No. 3, August 1979, pp. 213 – 225.

Appendix A

In order to estimate the variance of the channel amplitude estimate we need the second order statistics

$$\begin{aligned} E\left\{H_{lqkp}\cdot H^{*}_{l'q'k'p'}\right\}_{lq\neq kp} &= \sum_{m}\sum_{m'} E\left\{C_{lkqp}[m]\cdot C^{*}_{l'k'q'p'}[m']\right\}\cdot E\left\{I_{lk}[m]\cdot I_{l'k'}[m']\right\} \\ &= \sum_{m}\sum_{m'}\frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\delta_{mm'}\frac{N_{lkqp}[m']}{N_l}\cdot E\left\{I_{lk}[m]\cdot I_{l'k'}[m']\right\} \\ &= \frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\sum_{m'}\frac{N_{lkqp}[m']}{N_l}E\left\{I^2_{lk}[m']\right\} \end{aligned} \tag{A1}$$

where we have used

$$E\left\{C_{lkqp}[m]\cdot C^{*}_{l'k'q'p'}[m']\right\}\cong\frac{1}{N_l}\cdot\delta_{ll'}\cdot\delta_{kk'}\cdot\delta_{qq'}\cdot\delta_{pp'}\cdot\delta_{mm'}\cdot\frac{N_{lkqp}[m']}{N_l} \tag{A2}$$

which is derived in Appendix B assuming random codes. In order to evaluate $E\left\{I^2_{lk}[m']\right\}$ we consider two cases: 1) $k=l$, and 2) $k\neq l$. For $k=l$ we have

$$
\begin{aligned}
E\{I_{ll}^2[m']\} &= \frac{1}{M^2} E\left\{\sum_{m=1}^{M}\sum_{n=1}^{M} b_l[m]\cdot b_l[n]\cdot b_l[m-m']\cdot b_l[n-m']\right\} \\
&= \frac{1}{M^2}\sum_{m=1}^{M}\sum_{n=1}^{M}[\delta_{m'0}+(1-\delta_{m'0})\delta_{mn}] \\
&= \delta_{m'0}+(1-\delta_{m'0})\frac{1}{M} \\
&= \delta_{m'0}\left(1-\frac{1}{M}\right)+\frac{1}{M}
\end{aligned}
\tag{A3}
$$

whereas for $k \neq l$ we have

$$
\begin{aligned}
E\{I_{lk}^2[m']\} &= \frac{1}{M^2} E\left\{\sum_{m=1}^{M}\sum_{n=1}^{M} b_l[m]\cdot b_l[n]\cdot b_k[m-m']\cdot b_k[n-m']\right\} \\
&= \frac{1}{M^2}\sum_{m=1}^{M}\sum_{n=1}^{M}\delta_{mn}\cdot\delta_{mn} \\
&= \frac{1}{M}
\end{aligned}
\tag{A4}
$$

Hence, combining Equations (A3) and (A4) we have

$$
E\{I_{lk}^2[m']\} = \delta_{kl}\cdot\delta_{m'0}\left(1-\frac{1}{M}\right)+\frac{1}{M} \tag{A5}
$$

Equation (A1) then becomes

$$
\begin{aligned}
E\{H_{lqkp}\cdot H_{l'q'k'p'}^{*}\}_{lq\neq kp} &= \frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\sum_{m'}\frac{N_{lkqp}[m']}{N_l}E\{I_{lk}^2[m']\} \\
&= \frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\sum_{m'}\frac{N_{lkqp}[m']}{N_l}\left\{\delta_{kl}\cdot\delta_{m'0}\left(1-\frac{1}{M}\right)+\frac{1}{M}\right\} \\
&= \frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\left\{\delta_{kl}\cdot\frac{N_{lkqp}[0]}{N_l}\left(1-\frac{1}{M}\right)+\frac{1}{M}\right\}
\end{aligned}
\tag{A6}
$$

Now specializing Equation (A6) to the case where $k = l$

$$
E\{|H_{lqlp}|^2\}_{q\neq p} = \frac{1}{N_l}\left\{\frac{N_{llqp}[0]}{N_l}\left(1-\frac{1}{M}\right)+\frac{1}{M}\right\} \tag{A7}
$$

The above expression is further, simplified if we assume that users are approximately synchronous so that $N_{llqp}[0] \sim N_l$, which gives

$$
E\{|H_{lqlp}|^2\}_{q\neq p} \cong \frac{1}{N_l} \tag{A8}
$$

Similarly, specializing Equation (A6) to the case where $k \neq l$

$$E\left\{|H_{lqkp}|^2\right\}_{l\neq k} = \frac{1}{MN_l} \tag{A9}$$

Appendix B

In Appendix A we used the approximation

$$E\left\{C_{lkqp}[m]\cdot C^*_{l'k'q'p'}[m']\right\} \equiv \frac{1}{N_l}\delta_{ll'}\delta_{kk'}\delta_{qq'}\delta_{pp'}\delta_{mm'}\cdot\frac{N_{lkqp}[m']}{N_l} \tag{B1}$$

under the restriction that $lq \neq kp$. We show here that this expression is exactly true for chip-synchronous users, and that the approximation is reasonably valid for chip-asynchronous users, particularly when differences in delay lag are greater than about 2 chips. The analysis is based on random user codes.

The user correlations can be explicitly related to the code correlations as follows

$$\begin{aligned} C_{lkqp}[m] &= \frac{1}{2N_l}\sum_i\sum_j g[(i-j)N_c + mT + \tau_{lq} - \tau_{kp}]\cdot c_l^*[i]\cdot c_k[j] \\ &= C_{lk}[\tau_{lkqp}[m]] \\ C_{lk}[\tau] &\equiv \frac{1}{2N_l}\sum_i\sum_j g[(i-j)N_c + \tau]\cdot c_l^*[i]\cdot c_k[j] \\ \tau_{lkqp}[m] &\equiv mT + \tau_{lq} - \tau_{kp} \end{aligned} \tag{B2}$$

Consider two cases: 1) $l \neq k$, and 2) $l = k$.

<u>Case 1</u>

When $l \neq k$ the second-order statistics become

$$\begin{aligned} E\left\{C_{lk}[\tau]\cdot C^*_{l'k'}[\tau']\right\} &= \frac{1}{4N_lN_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\left\{c_l^*[i]\cdot c_{l'}[i']\cdot c_k[j]\cdot c^*_{k'}[j']\right\} \\ &= \frac{1}{4N_lN_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\delta_{ll'}\delta_{ii'}\cdot 2\delta_{kk'}\delta_{jj'} \\ &= \frac{\delta_{ll'}\cdot\delta_{kk'}}{N_l^2}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] \\ g_{ij}[\tau] &\equiv g[(i-j)N_c + \tau] \end{aligned} \tag{B3}$$

where we have used the assumption of random user codes, independent among the users. Note also that the summation over i is over the range where $c_l[i]$ is non-zero, and similarly the summation over j is over the range where $c_k[j]$ is non-zero.

Case 2

Now consider case 2 where $l = k$

$$\begin{aligned} E\{C_{ll}[\tau]\cdot C^*_{l'k'}[\tau']\} &= \frac{1}{4N_l N_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_{l'}[i']\cdot c_l[j]\cdot c^*_{k'}[j']\} \\ &= \frac{\delta_{l'k'}}{4N_l N_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[j]\cdot c_{l'}[i']\cdot c^*_{l'}[j']\} \end{aligned} \tag{B4}$$

When $l \neq l'$ we have

$$\begin{aligned} E\{C_{ll}[\tau]\cdot C^*_{l'k'}[\tau']\} &= \frac{\delta_{l'k'}}{4N_l N_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[j]\cdot c_{l'}[i']\cdot c^*_{l'}[j']\} \\ &= \frac{\delta_{l'k'}}{4N_l N_{l'}}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\delta_{ij}\cdot 2\delta_{i'j'} \\ &= \frac{\delta_{l'k'}}{N_l N_{l'}}\sum_{ii'} g[\tau]\cdot g[\tau'] \\ &= \delta_{l'k'} g[\tau]\cdot g[\tau'] \end{aligned} \tag{B5}$$

whereas when $l = l'$ we have

$$\begin{aligned} E\{C_{ll}[\tau]\cdot C^*_{lk'}[\tau']\} &= \frac{1}{4N_l^2}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[i']\cdot c_l[j]\cdot c^*_{k'}[j']\} \\ &= \frac{\delta_{lk'}}{4N_l^2}\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[i']\cdot c_l[j]\cdot c^*_l[j']\} \\ &= \frac{\delta_{lk'}}{4N_l^2}\left\{\sum_{i\neq j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[i']\cdot c_l[j]\cdot c^*_l[j']\}\right. \\ &\qquad \left. + \sum_{i=j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot E\{c^*_l[i]\cdot c_l[i']\cdot c_l[j]\cdot c^*_l[j']\}\right\} \end{aligned} \tag{B6a}$$

$$\begin{aligned} &= \frac{\delta_{lk'}}{4N_l^2}\left\{\sum_{i\neq j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\delta_{ii'}\cdot 2\delta_{jj'}\right. \\ &\qquad \left. + \sum_{i=j,i'j'} g[\tau]\cdot g_{i'j'}[\tau']\cdot 2E\{c_l[i']\cdot c^*_l[j']\}\right\} \\ &= \frac{\delta_{lk'}}{4N_l^2}\left\{\sum_{iji'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\delta_{ii'}\cdot 2\delta_{jj'} - \sum_{i=j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\delta_{ii'}\cdot 2\delta_{jj'}\right. \\ &\qquad \left. + \sum_{i=j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2E\{c_l[i']\cdot c^*_l[j']\}\right\} \end{aligned} \tag{B6b}$$

$$
\begin{aligned}
&= \frac{\delta_{lk'}}{4N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau']\cdot 2\cdot 2 - \sum_i g[\tau]\cdot g[\tau']\cdot 2\cdot 2\right.\\
&\qquad\left. + \sum_{i=j,i'j'} g_{ij}[\tau]\cdot g_{i'j'}[\tau']\cdot 2\cdot 2\delta_{i'j'}\right\}\\
&= \frac{\delta_{l'k'}}{N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - N_l g[\tau]\cdot g[\tau'] + N_l^2 g[\tau]\cdot g[\tau']\right\}\\
&= \delta_{l'k'} g[\tau]\cdot g[\tau'] + \frac{\delta_{l'k'}}{N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - N_l g[\tau]\cdot g[\tau']\right\}
\end{aligned}
\tag{B6c}
$$

Hence combining Equations (B5) and (B6c) we have

$$
E\{C_{ll}[\tau]\cdot C^*_{l'k'}[\tau']\} = \delta_{l'k'} g[\tau]\cdot g[\tau'] + \frac{\delta_{ll'}\cdot\delta_{l'k'}}{N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - N_l g[\tau]\cdot g[\tau']\right\}
\tag{B7}
$$

and combining cases for $l \neq k$ and $l = k$ we have

$$
\begin{aligned}
E\{C_{lk}[\tau]\cdot C^*_{l'k'}[\tau']\} &= \delta_{lk}\cdot\delta_{l'k'} g[\tau]\cdot g[\tau'] + \frac{\delta_{lk}\cdot\delta_{ll'}\cdot\delta_{l'k'}}{N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - N_l g[\tau]\cdot g[\tau']\right\}\\
&\quad + (1-\delta_{lk})\frac{\delta_{ll'}\cdot\delta_{kk'}}{N_l^2}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau']\\
&= \delta_{lk}\cdot\delta_{l'k'} g[\tau]\cdot g[\tau'] + \frac{\delta_{lk}\cdot\delta_{ll'}\cdot\delta_{l'k'}}{N_l^2}\left\{\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - N_l g[\tau]\cdot g[\tau']\right\}\\
&\quad + \frac{\delta_{ll'}\cdot\delta_{kk'}}{N_l^2}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] - \frac{\delta_{lk}\cdot\delta_{ll'}\cdot\delta_{l'k'}}{N_l^2}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau']\\
&= \delta_{lk}\cdot\delta_{l'k'} g[\tau]\cdot g[\tau'] - \frac{\delta_{lk}\cdot\delta_{ll'}\cdot\delta_{l'k'}}{N_l} g[\tau]\cdot g[\tau'] + \frac{\delta_{ll'}\cdot\delta_{kk'}}{N_l^2}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau']
\end{aligned}
\tag{B8}
$$

The above expression can be used to determine the second-order statistics for the general case of symbol-asynchronous and chip-asynchronous users with arbitrary spreading factors. In what follows we will be interested in approximating the above expression so as to get simple but meaningful results. In order to simplify the expressions we consider users all at the highest spreading factor, and we assume that certain small values are zero.

To assess the accuracy of channel estimation we need to determine the second order statistics

$$
\begin{aligned}
&E\{C_{lkqp}[m]\cdot C^*_{l'k'q'p'}[m']\} = E\{C_{lk}[\tau_{lkqp}[m]]\cdot C^*_{l'k'}[\tau_{l'k'q'p'}[m']]\}\\
&\tau_{lkqp}[m] \equiv mT + \tau_{lq} - \tau_{kp}
\end{aligned}
\tag{B9}
$$

with $lq \neq kp$. The function $g[\tau]g[\tau']$ in Equation (B8) above is small unless both τ and τ' are close to zero, and for the chip-asynchronous case function is exactly zero since unless both τ and τ' are equal to zero. Since for $lq \neq kp$ the probability that $\tau_{lkqp}[m]$ is close to zero is small a good approximation is to assume that these functions are zero. The third term can be written

$$E\{C_{lk}[\tau]\cdot C^{*}_{l'k'}[\tau']\} \cong \frac{\delta_{ll'}\delta_{kk'}}{N_l}\left\{\frac{1}{N_l}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau']\right\} \tag{B10}$$

The double summation in the brackets

$$S_{lk}[\tau,\tau'] \equiv \frac{1}{N_l}\sum_{ij} g_{ij}[\tau]\cdot g_{ij}[\tau'] \tag{B11}$$

is plotted in Figure B1 for $N_l = N_k = 256$ versus $\tau - \tau'$ for $(\tau + \tau')/2 = 0$.

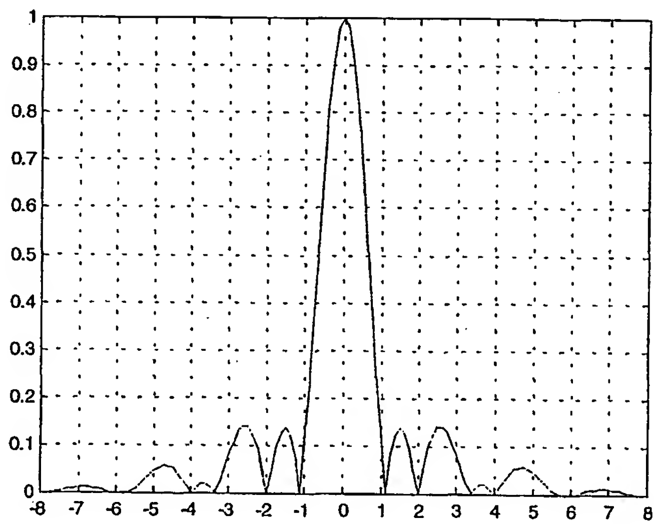


Figure B1. Plot of $S_{lk}[\tau,\tau']$ for $N_l = N_k = 256$ versus $\tau - \tau'$ for $(\tau + \tau')/2 = 0$.

The sharp localization around $\tau - \tau' = 0$ is valid for all values of $(\tau + \tau')/2$, except that for $(\tau + \tau')/2$ large peak value drops off due to the partial overlap of the codes. Hence for delay lag differences $\tau - \tau'$ greater than about 2 chips a good approximation is

$$S_{lk}[\tau,\tau'] \cong \delta_{\tau\tau'}\cdot S_{lk}[\tau,\tau] \tag{B12}$$

This approximation then gives

$$E\{C_{lk}[\tau]\cdot C^{*}_{l'k'}[\tau']\} \cong \frac{\delta_{ll'}\cdot\delta_{kk'}}{N_l}\cdot\delta_{\tau\tau'}\cdot S_{lk}[\tau,\tau] \tag{B13}$$

which implies

$$E\{C_{lkqp}[m]\cdot C^{*}_{l'k'q'p'}[m']\}\cong\frac{1}{N_l}\cdot\delta_{ll'}\cdot\delta_{kk'}\cdot\delta_{qq'}\cdot\delta_{pp'}\cdot\delta_{mm'}\cdot S_{lk}[\tau,\tau] \tag{B14}$$

provided the delay spread is less than a symbol period. Now it can be shown that

$$\begin{aligned} S_{lk}[\tau_{lkqp}[m'],\tau_{lkqp}[m']] &= \frac{1}{N_l}\sum_{ij} g_{ij}^{2}[\tau_{lkqp}[m']] \\ &\cong \frac{N_{lkqp}[m']}{N_l} \end{aligned} \tag{B15}$$

where $N_{lkqp}[m']$ is the overlap between the user codes. Our final result is then

$$E\{C_{lkqp}[m]\cdot C^{*}_{l'k'q'p'}[m']\}\cong\frac{1}{N_l}\cdot\delta_{ll'}\cdot\delta_{kk'}\cdot\delta_{qq'}\cdot\delta_{pp'}\cdot\delta_{mm'}\cdot\frac{N_{lkqp}[m']}{N_l} \tag{B16}$$

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: MUD interface to modem Date: January 3, 2001

1. Multi-User Signal Model

The Rake receiver operation described in the next section is based a signal model. The MUD algorithm and implementation are based on the same model. This model is described below.

Figure 1 shows how the uplink complex spreading for the Dedicated Physical Data CHannels (DPDCHs) and the Dedicated Physical Control CHannel (DPCCH). There can be from 1 to 6 DPDCHs, denoted $DPDCH_k$, for k from 1 to 6. If there is more than one DPDCH, then the spreading factor for all DPDCHs must be equal to 4. For a single DPDCH ($DPDCH_1$) the spreading factor can vary from 4 to 256. The data bits for channel $DPDCH_1$ are spread by channelization code $c_{d,1} = C_{ch,SF,SF/4}$, where SF is the DPDCH spreading factor. These channelization codes are referred to as Orthogonal Variable Spreading Factor (OVSF) codes. They are equivalent to Hadamard codes, except for their ordering. When there are multiple DPDCHs then dedicated channels $DPDCH_k$, for k from 1 to 6 are spread by channelization codes $c_{d,k} = C_{ch,4,n}$, where the relationship between n and k is represented in Table 1.

Table 1. Relationship between n and k.

n	*k*
1	*1,2*
3	*3,4*
2	*5,6*

The data bits for the DPCCH are spread by code $c_c = C_{ch,256,0}$. The spreading factor for the DPCCH is always equal to 256. The multipliers β_c and β_d are constants used to select the relative amplitudes of the control and data channels. At least one of these constants must be equal to 1 for any given symbol period m.

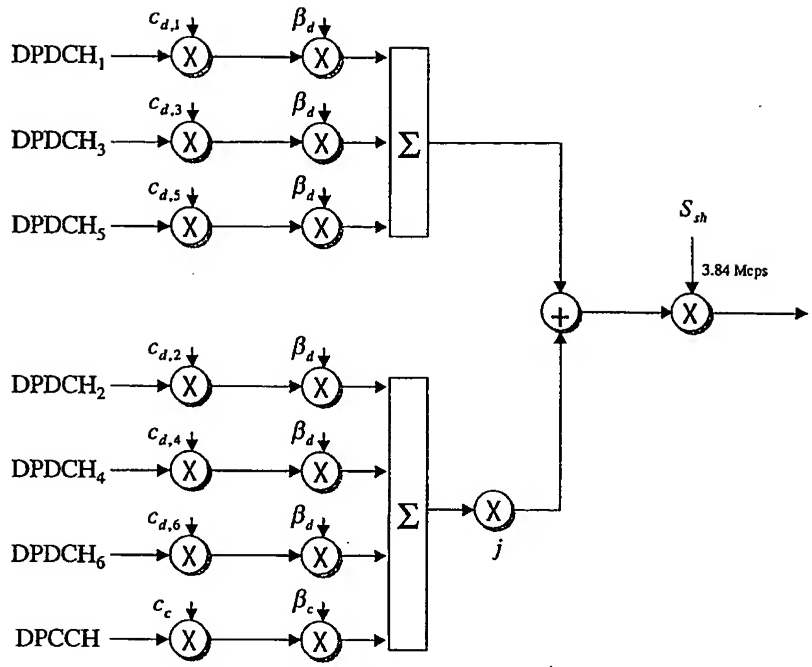


Figure 1. Uplink complex spreading of DPDCHs and DPCCH

The uplink spreading for any one of the seven Dedicated CHannels (DCHs) above can be represented as shown in Figure 2.

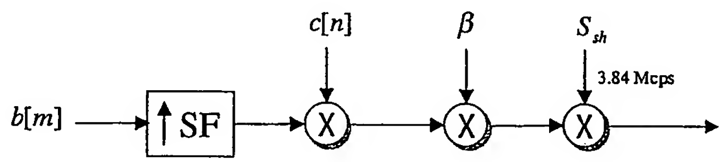


Figure 2. A second representation of the uplink spreading for any one of the seven Dedicated CHannels (DCHs).

where the code *c[n]* is given by

$$c[n]=\begin{cases} C_{ch,256,0}[n]\cdot jS_{sh}[n], & \text{DPCCH} \\ C_{ch,256,64}[n]\cdot S_{sh}[n], & \text{DPDCH}_1 \\ C_{ch,256,64}[n]\cdot jS_{sh}[n], & \text{DPDCH}_2 \\ C_{ch,256,192}[n]\cdot S_{sh}[n], & \text{DPDCH}_3 \\ C_{ch,256,192}[n]\cdot jS_{sh}[n], & \text{DPDCH}_4 \\ C_{ch,256,128}[n]\cdot S_{sh}[n], & \text{DPDCH}_5 \\ C_{ch,256,128}[n]\cdot jS_{sh}[n], & \text{DPDCH}_6 \end{cases} \tag{1}$$

and

$$\beta = \begin{cases} \beta_c, & \text{DPCCH} \\ \beta_d, & \text{DPDCH}_{1-6} \end{cases} \qquad (2)$$

For a DCH with a spreading factor less than 256 there are $J = 256/SF$ data bits transmitted during a single 256-chip symbol period (i.e. 1/15 ms). From a signal model perspective, the J data bits transmitted per symbol period can be viewed as arising from J *virtual users*, each transmitting a single bit per symbol period. The idea is illustrated in Figure 3.

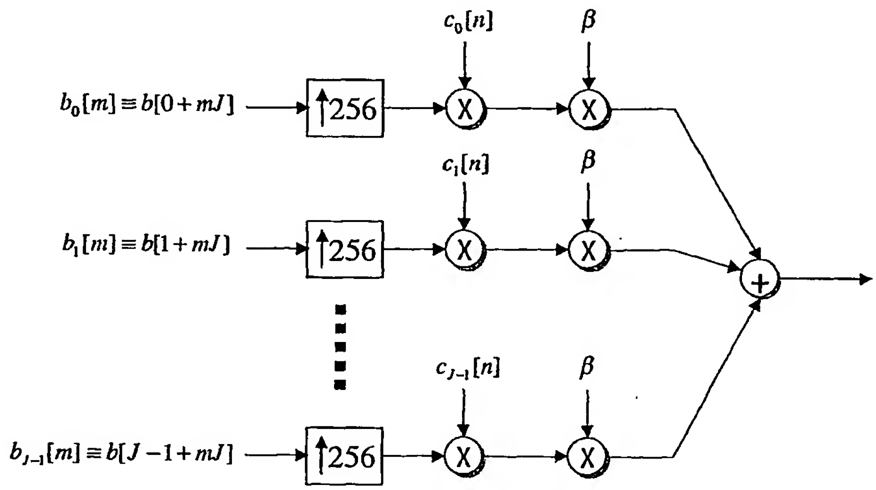


Figure 3. Transforming a single user with bit rate J bits per symbol period into J virtual users, each with bit rate 1 bit per symbol period.

The codes for these virtual users are formed by extracting SF elements at a time out of the DCH code sequence to form J new codes. Each of the J codes is of length 256 chips, but with only SF non-zero chips. That is,

$$c_j[n] \equiv \begin{cases} c[n], & j \cdot SF \le n < (j+1) \cdot SF \\ 0, & \text{otherwise} \end{cases} \qquad (3)$$

This code-partitioning concept is illustrated in Figure 4 for the case $SF = 64$ so that $J = 256/SF = 4$ codes are derived from the one DCH code.

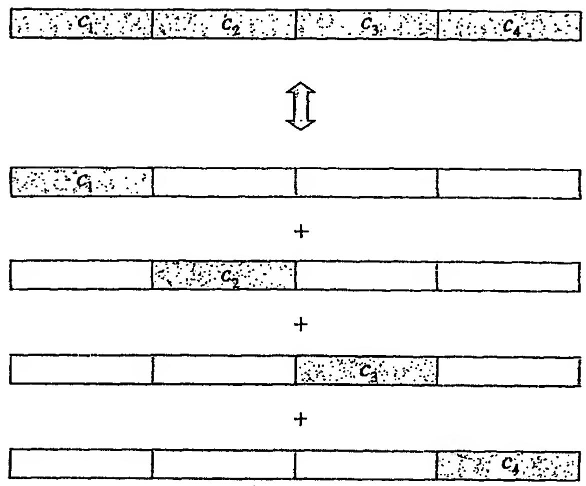


Figure 4. Code partitioning concept illustrated for the case SF = 64, whereby J = 256/SF = 4 codes are derived from a single DCH code.

The control channel can also be viewed as a virtual user. Hence, for a given physical user with spreading factor SF there are $1 + 256N_D/SF$ virtual users, where N_D is the number of DPDCHs. (Recall that for $N_D > 1$, $SF = 4$.)

It turns out to be convenient to use a double indexing scheme to i dentify virtual users. Let paired indices kj represent the jth virtual user associated with the kth dedicated channel. Index j varies from $0 <= j < J_k = 256/SF_k$, where SF_k is the spreading factor for the kth dedicated channel. For the remainder of this section the spreading factors SF_k are assumed to be constant. In section 3 the equations are reformulated to allow for symbol-by-symbol changes in the spreading factor.

The transmitted signal for virtual user kj can be written

$$x_{kj}[t] = \beta_k \sum_m v_{kj}[t - mT] b_{kj}[m] \tag{4}$$

where t is the integer time sample index, $T = NN_c$ is the data bit duration, $N = 256$ is the short-code length, N_c is the number of samples per chip, $b_{kj}[m]$ are the data bits, and where $v_{kj}[t]$ is the transmit signature waveform for virtual user kj. This waveform is generated by passing the spread code sequence $c_{kj}[n]$ through a root-raised-cosine pulse-shaping filter $h[t]$

$$s_{kj}[t] = \sum_{p=0}^{N-1} h[t - pN_c] c_{kj}[p] \tag{5}$$

Note that $\beta_k = \beta_c$ if the kjth virtual user corresponds to a control channel. Otherwise $\beta_k = \beta_d$.

The total number of virtual users is denoted

$$K_v \equiv \sum_{k=1}^{K_D} \frac{256}{SF_k} \tag{6}$$

where K_D is the total number of dedicated channels. The baseband received signal after root-raised-cosine matched-filtering can be written

$$r[t] = \sum_{k=1}^{K_D} \sum_{j=0}^{J_k-1} \sum_{m} \tilde{s}_{kj}[t-mT]b_{kj}[m] + w[t] \tag{7}$$

where $w[t]$ is receiver noise with a raised-cosine power spectral density, and where $\tilde{s}_{kj}[t]$ is the channel-corrupted signature waveform for virtual user kj. For L multipath components the channel-corrupted signature waveform for virtual user kj is modeled as

$$\tilde{s}_{kj}[t] = \sum_{p=1}^{L} a_{kp} s_{kj}[t-\tau_{kp}] \tag{8}$$

where a_{kp} are the complex multipath amplitudes. The amplitude ratios β_k are incorporated into the amplitudes a_{kp}. Notice that if k and l are two dedicated channels corresponding to the same physical user then, aside from scaling the by β_k and β_l, a_{kp} and a_{lp}, are equal. This is due to the fact that the signal waveforms of all virtual users corresponding to the same physical user pass through the same channel. The waveform $s_{kj}[t]$ is referred to as the signature waveform for the $kjth$ virtual user. This waveform is generated by passing the spread code sequence $c_{kj}[n]$ through a raised cosine pulse-shaping filter $g[t]$

$$s_{kj}[t] = \sum_{p=0}^{N-1} g[t-pN_c]c_{kj}[p] \tag{9}$$

Note that for spreading factors less than *256* some of the chips $c_{kj}[p]$ are zero.

2. Rake Receiver Operation

This section describes the operation of a typical Rake receiver. Figure 1 shows a representation of the received antenna data that is delivered to the Rake receivers of all users.

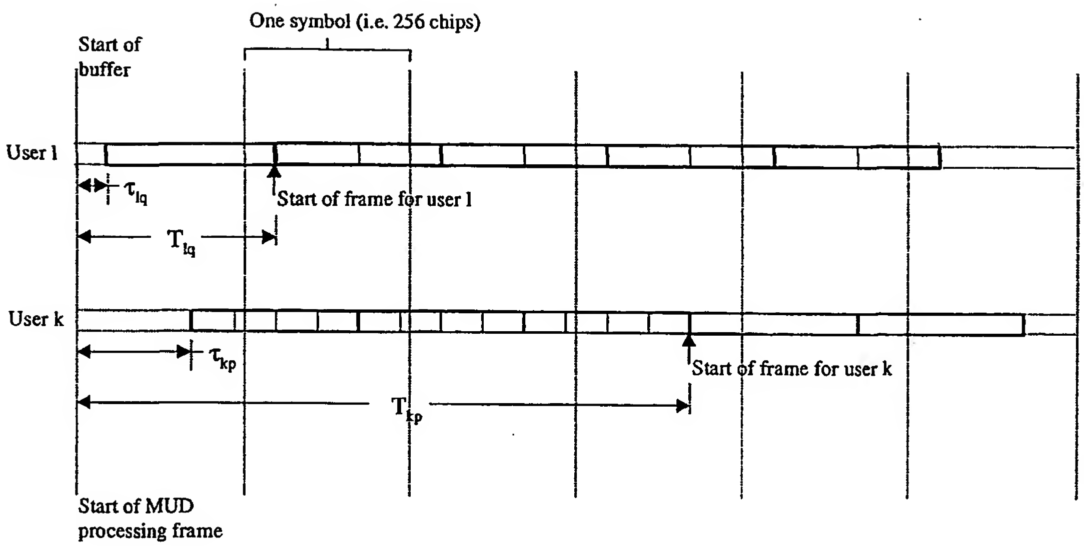


Figure 5. Received antenna data delivered to the Rake receivers of all users.

The figure shows the received signals corresponding to users *l* and *k*. These signals are combined in free space so that the receivers gets one composite signal, which we denote *r[t]*. The buffer length is assumed to be an integral number of frames in length so that delay lag values T_{lq} are approximately constant with each new filling of the buffer. For each finger of each user there is a delay lag value T_{lq} indicating the start of frame for the *q*th multipath of the *l*th user. Lag values T_{lq} are assumed to be constant over a frame, but are allowed to change from frame to frame in response to the delay locked loop operation and in response to new searcher-receiver sweeps where new delay lags are found. The lower case values $\tau_{lq} = T_{lq} \bmod 256N_c$ denote the symbol-period offset relative to the start of an internal symbol period reference clock. Notice that the user spreading factors change on user frame boundaries. Since users are asynchronous it is impossible to have a MUD processing frame that corresponds to all user frame boundaries. Hence the MUD processing frame is matched as close as possible to the user frame boundaries, but does not necessarily correspond precisely to any user's frame boundary. Consequently there will be spreading factor changes that occur during a MUD processing frame. Handling these mid-frame changes is the subject of section 3 below.

The received signal above, which has been match-filtered to the chip pulse, must next be match-filtered by the user code-sequence filter. Since the spreading factor for the DPDCHs is not known, the Rake receiver performs an initial 4-chip despreading over all DPDCHs. The Fast Hadamard Transformation (FHT) can be used here to reduce the number of operations. The detection statistics for the multiple fingers and multiple antennas are maximal-ratio combined. Since the DPCCH is always spread with a spreading factor of 256 the DPCCH can be entirely despread during each symbol period. TFCI bits are extracted each slot from the DPCCH. After an entire frame is processed the TFCI is decoded and the spreading factor for that frame is determined. After spreading factor determination the final DPDCH despreading is performed. The resulting detection statistics are denoted here as $y_{kj}[m]$, the matched-filter output for the *kj*th virtual user for the *m*th symbol period. Since there are K_v codes, there are K_v such detection statistics,

which are collected into a column vector $y[m]$ for the mth symbol period. The matched-filter output $y_{li}[m]$, for the $l$ith virtual user can be written

$$y_{li}[m] = \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot y_{li,q}[m]\right\}$$
$$y_{li,q}[m] \equiv \frac{1}{2N_l}\sum_{n} r[nN_c + \hat{\tau}_{lq} + mT] \cdot c_{li}^{*}[n] \tag{10}$$

where $\hat{a}_{lq}$ is the estimate of a_{lq}, $\hat{\tau}_{lq}$ is the estimate of τ_{lq}, and N_l is the (non-zero) length of codes $c_{li}[n]$ (i.e., the spreading factor for the lth dedicated channel). The intermediate result $y_{li,q}[m]$ represents the despread signal at the qth lag, and is here referred to the pre-MRC matched-filter output. When multiple antennas are employed, $r[t]$, $y_{li,q}[m]$ and $\hat{a}_{lq}$ are column vectors with one complex element per antenna.

The matched-filter detector estimates the transmitted data bits as $\hat{b}_{li}[m] \equiv sign\{y_{li}[m]\}$. Multiuser detection is considered in the next section.

3. Multiuser Detection Equations and Asynchronous Processing

As shown in Figure 5 a MUD processing interval must necessarily by asynchronous with most user's frame boundaries since the users are asynchronous. Because of this spreading factors will change during a MUD processing frame. When the spreading factor changes during the processing frame the MUD equations are modified. These modifications are considered in this section.

The modem delivers matched-filter data to the MUD function on a frame-by-frame basis. Let $N_P[r]$ represent the number of physical users accessing the system during frame r. For each frame the following data is received for physical users p = 1 to $N_P[r]$ and each dedicated channel l

- Number of DPDCHs, $N_{D,p}$
- Spreading factor, SF_l
- Amplitude ratios β_d and β_c
- Slot format
- Channel amplitude estimates a_{lq}
- Channel lag estimates T_{lq}
- Matched-filter outputs $f_{li}[m]$ for all DCHs
- Code numbers
- Gap information for compressed mode

Matched-filter outputs $f_{li}[m]$ correspond to the matched-filter outputs $y_{li}[m]$. If the lth dedicated channel is a DPCCH then matched-filter outputs are only received for the TPC, TFCI and FBI bits. The $f_{li}[m]$ values are mapped to the $y_{li}[m]$ values as described below. The mapping accounts for the frame offsets between the various users. The amount of matched-filter data received per physical user depends on the DPDCH spreading factor.

For each dedicated channel a symbol offset m_l is determined according to

$$m_l \equiv \left\{ \frac{1}{L} \sum_{q=1}^{L} T_{lq} \right\} \text{div } (256 N_c) \tag{11}$$

where *div* denotes integer division (i.e. with truncation). The symbol offset represents the fact that the users and hence the frame data are asynchronous. The y-data used for interference cancellation is derived from the frame data using

$$y_{li}[m] = f_{li}[m - m_l] \tag{12}$$

Figure 6 shows an example mapping of user data frames to MUD processing frames. To illustrate concepts the frames are each 16 symbol periods long rather than the actual 150 symbols for WCDMA. The height of the blocks represents the number of virtual users per physical user. For physical users 1 and 4 the spreading factor changes in going from data frame 1 to data frame 2. As shown in the figure this results in spreading factor changes within the MUD processing frame. The MUD function is designed to Calculate the C-matrix once per frame. Hence mid-frame changes to user spreading factors pose a problem which requires special treatment. It turns out, and will be shown below, that mid-frame changes to the spreading factor can be accommodated by performing modified calculations based on the minimum spreading factor over the MUD processing frame.

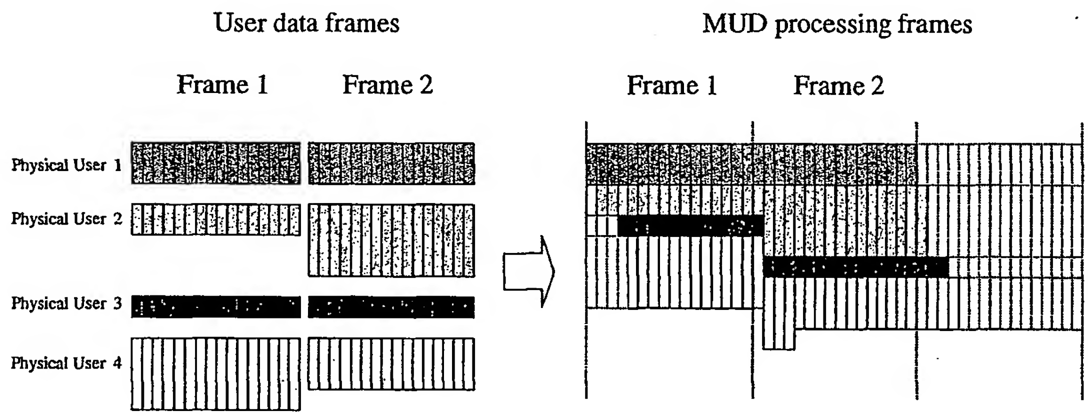


Figure 6. Mapping of user data frames to MUD processing frames.

First we develop the MUD matrix signal model which allows user spreading factors to change on a symbol-by-symbol basis. We then show how we can perform the processing based on the minimum user spreading factors over the MUD processing frame.

Let us reformulate the signal model presented in section 1 so as to allow spreading factors to change every symbol period. For every DCH k, there are $J_k[m]$ virtual users, where index m is the symbol period index. The number of DCHs $J_k[m]$ is

$$J_k[m] \equiv \frac{256}{SF_k[m]} \tag{13}$$

where $SF_k[m]$ is the spreading factor for the kth dedicated channel during the mth symbol period. The signature waveform for the jth virtual user of $J_k[m]$ total belonging to the kth DCH over the mth symbol period can be written

$$s_{kj,m}[t] = \sum_{p=0}^{N-1} g[t - pN_c] c_{kj,m}[p] \tag{14}$$

where the codes and hence the signature waveforms now include the symbol-period index m to account for symbol-by-symbol spreading factor changes. The channel-corrupted signature waveform is then

$$\tilde{s}_{kj,m}[t] = \sum_{p=1}^{L} a_{kp} s_{kj,m}[t - \tau_{kp}] \tag{15}$$

and thus the received signal corresponding to K_D dedicated channels is

$$r[t] = \sum_{k=1}^{K_D} \sum_{m} \sum_{j=0}^{J_k[m]-1} \tilde{s}_{kj,m}[t - mT] b_{kj}[m] + w[t] \tag{16}$$

The MUD matrix signal model proceeds from substituting the received signal $r[t]$ from Equation (16) into Equation (10) for the matched-filter outputs

$$y_{li}[m] \equiv \sum_{n} \sum_{k=1}^{K_D} \sum_{j=0}^{J_k[n]-1} \mathrm{Re}\left\{ \sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l[m]} \sum_{r} \tilde{s}_{kj,n}[rN_c + \hat{\tau}_{lq} + (m-n)T] \cdot c_{li,m}^*[r] \right\} b_{kj}[n] + \eta_{li}[m]$$

$$= \sum_{n} \sum_{k=1}^{K_d} \sum_{j=0}^{J_k[n]-1} r_{likj}[m,n] b_{kj}[n] + \eta_{li}[m]$$

$$r_{likj}[m,n] = \sum_{q=1}^{L} \sum_{p=1}^{L} \mathrm{Re}\left\{ \hat{a}_{lq}^H a_{kp} \cdot C_{likjqp}[m,n] \right\} \tag{17}$$

$$C_{likjqp}[m,n] \equiv \frac{1}{2N_l[m]} \sum_{r} \sum_{s} g[(r-s)N_c + (m-n)T + \hat{\tau}_{lq} - \tau_{kp}] c_{li,m}^*[r] \cdot c_{kj,n}[s]$$

where $\eta_{li}[m]$ is the match-filtered receiver noise and $N_l[m] = SF_l[m]$. The terms for $m' \neq 0$ result from asynchronous users.

The delay lags τ_{lq} for a given DCH l will under most circumstances be grouped within a range of from 4 to 8 μs. Under extreme conditions the delay spread will be as high as 20 μs. In any event, let τ_l represent the mean delay lag τ_{lq} over index q. According to Equation (10) above, the matched-filter detection statistic $y_{li}[0]$ is the result found by correlating the received signal starting roughly at delay lag τ_l, where τ_l is approximately in the range 0 to $256N_c$. If τ_l moves significantly outside this range an adjustment in the symbol period alignment will need to be made to restore τ_l back to within the desired range. More will be said about this below. Along the same lines, the detection statistic

$y_{li}[m]$ is the result found by correlating the received signal starting roughly at delay lag $\tau_l + mT$.

For efficient MUD processing it is important for the C-matrices to be constant over a 10 ms MUD processing frame. We now describe a method which operates on constant C-matrices. Handling changes to user spreading factors is relegated to the IC portion of the MUD processing. Let us define

$$\mathbf{J}_k \equiv \max_m J_k[m] \tag{18}$$

where the maximization is over symbol periods m that contribute to the current MUD processing frame. This includes not only symbol periods that fall within the MUD processing frame, but in addition a few symbol periods on either side due to asynchronous users. Note that the minimum spreading factor for the kth DCH is $SF_k = 256/\mathbf{J}_k$. Now define the DCH *contraction factor* for the mth symbol period as

$$C_k[m] \equiv \frac{\mathbf{J}_k}{J_k[m]} \tag{19}$$

The DCH codes for a given symbol period can be expressed as a sum of the DCH codes corresponding to the minimum spreading factor. For the kth DCH there are at most $\mathbf{J}_k$ virtual users corresponding to the minimum spreading factor. Let the codes for these users be denoted $\mathbf{c}_{kj}[r]$, $0 <= j < \mathbf{J}_k$. The codes for the mth symbol period, where there might be fewer virtual users, are denoted $c_{kj,m}[r]$, $0 <= j < J_k[m]$, where

$$c_{kj,m}[r] = \sum_{j'=j \cdot C_k[m]}^{(j+1)\cdot C_k[m]-1} \mathbf{c}_{kj'}[r] \tag{20}$$

With this result we are now able to represent the MUD signal model in terms of the C-matrix and R-matrix elements based on the codes corresponding to the minimum DCH spreading factors. The C-matrix in Equation () above becomes

$$\begin{aligned} C_{likjqp}[m,n] &\equiv \frac{1}{2N_l[m]} \sum_r \sum_s g[(r-s)N_c + (m-n)T + \hat{\tau}_{lq} - \tau_{kp}] c^*_{li,m}[r] \cdot c_{kj,n}[s] \\ &= \frac{1}{2N_l[m]} \sum_r \sum_s g[(r-s)N_c + (m-n)T + \hat{\tau}_{lq} - \tau_{kp}] \sum_{i'=i \cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \mathbf{c}^*_{li'}[r] \cdot \sum_{j'=j \cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{c}_{kj'}[s] \\ &= \frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i \cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j \cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \frac{1}{2\mathbf{N}_l} \sum_r \sum_s g[(r-s)N_c + (m-n)T + \hat{\tau}_{lq} - \tau_{kp}] \mathbf{c}^*_{li'}[r] \cdot \mathbf{c}_{kj'}[s] \\ &= \frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i \cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j \cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{C}_{li'kj'qp}[m-n] \end{aligned}$$

$$\mathbf{C}_{likjqp}[m-n] \equiv \frac{1}{2\mathbf{N}_l} \sum_r \sum_s g[(r-s)N_c + (m-n)T + \hat{\tau}_{lq} - \tau_{kp}] \mathbf{c}^*_{li}[r] \cdot \mathbf{c}_{kj}[s] \tag{21}$$

where $N_l = min\ N_l[m] = SF_l$. Similarly, the R-matrix becomes

$$\begin{aligned} r_{likj}[m,n] &= \sum_{q=1}^{L}\sum_{p=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{H} a_{kp} \cdot C_{likjqp}[m,n]\right\} \\ &= \frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i\cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \sum_{q=1}^{L}\sum_{p=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{H} a_{kp} \cdot \mathbf{C}_{li'kj'qp}[m-n]\right\} \\ &= \frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i\cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{r}_{li'kj'}[m-n] \end{aligned} \tag{22}$$

$$\mathbf{r}_{likj}[m-n] \equiv \sum_{q=1}^{L}\sum_{p=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{H} a_{kp} \cdot \mathbf{C}_{likjqp}[m-n]\right\}$$

so that the matched-filter outputs become

$$\begin{aligned} y_{li}[m] &\equiv \sum_{n}\sum_{k=1}^{K_D}\sum_{j=0}^{J_k[n]-1} r_{likj}[m,n] b_{kj}[n] + \eta_{li}[m] \\ &= \sum_{n}\sum_{k=1}^{K_D}\sum_{j=0}^{J_k[n]-1} \left\{\frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i\cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{r}_{li'kj'}[m-n]\right\} b_{kj}[n] + \eta_{li}[m] \end{aligned} \tag{23}$$

This last equation can be written

$$\begin{aligned} y_{li}[m] &= \sum_{n}\sum_{k=1}^{K_D}\sum_{j=0}^{J_k[n]-1} \left\{\frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i\cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{r}_{li'kj'}[m-n]\right\} b_{kj}[n] + \eta_{li}[m] \\ &= \frac{\mathbf{N}_l}{N_l[m]} \sum_{i'=i\cdot C_l[m]}^{(i+1)\cdot C_l[m]-1} \mathbf{y}_{li'}[m] + \eta_{li}[m] \end{aligned}$$

$$\begin{aligned} \mathbf{y}_{li'}[m] &\equiv \sum_{n}\sum_{k=1}^{K_D}\sum_{j=0}^{J_k[n]-1} \left\{\sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{r}_{li'kj'}[m-n]\right\} b_{kj}[n] \\ &= \sum_{n}\sum_{k=1}^{K_D}\sum_{j=0}^{J_k[n]-1} \left\{\sum_{j'=j\cdot C_k[n]}^{(j+1)\cdot C_k[n]-1} \mathbf{r}_{li'kj'}[m-n] \cdot \mathbf{b}_{kj'}[n]\right\} \\ &= \sum_{n}\sum_{k=1}^{K_D}\sum_{j'=0}^{\mathbf{J}_k-1} \mathbf{r}_{li'kj'}[m-n] \cdot \mathbf{b}_{kj'}[n] \end{aligned} \tag{24}$$

where we have defined $\mathbf{b}_{kj'}[n] = b_{kj}[n]$ for $jC_k[n] <= j' < (j+1)C_k[n]$. Equation (24) is based entirely in terms of matrix elements corresponding to the minimum spreading factor for the MUD processing frame.

Computer Systems Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: WCDMA Downlink MUD Date: February 23, 2001

1. Introduction

MultiUser Detection (MUD) is most often thought of as a technique to improve either capacity or coverage for the *uplink*. A few reasons why MUD is uplink-focussed are

- Downlink MUD must be performed in the handsets, which are limited in processing power
- Each handset is interested in only one signal
- In the downlink users are separated by orthogonal codes

However, there is typically a greater demand for capacity in the downlink. If MUD is only applied in the uplink the imbalance is even greater. While in the downlink users are separated by orthogonal codes, because of multipath there is still significant intra-cell interfernece. Equalization has been suggested as a means of restoring orthogonality, however the computationally attractive linear equalization methods tend to amplify the othe-cell interference and noise.

A downlink MUD method is described in the next section which has reduced complexity. The Fast Hadamard Transform (FHT) is used to reduce complxity. The FHT is used in both the forward (demodulation) and backward (regeneration) directions.

2. The Method

The method proceeds according to the following steps

- Receive amplitude and delay information form the searcher receiver
- Start with the largest multipath
- Multiply the received signal by the conjugate of the scrambling code (512 chips at a time)
- Perform the FHT on the result (for multirate users, this is done in stages)
- Determine soft data estimates

- Set user-of-interest data symbols to zero.
- Do same for all multipaths
- Proceed till end of slot
- Estimate amplitudes and gain factors
- Diversity combine results and make hard decisions
- Use hard decisions, gain estimates and FHT to reconstruct chip sequence *c[n]*.(with user of interest nulled)
- Multiple *c[n]* by $c_{sh}[n]$ to form *d[n]* (with user of interest nulled)
- Use amplitude estimates, delay lag estimates (from searcher) and raised-cosine pulse to construct chip filter
- Pass *d[n]* (with user of interest nulled) through chip filter to reconstruct interference signal
- Subtract interference signal from received signal
- Demodulate with conventional RAKE receiver

The WCDMA transmitted signal can be represented as

$$s[t] = \sum_n g[t - nN_c]d[n]$$

$$\begin{aligned} d[n] &= \left\{ \sum_{k=1}^{K} G_k b_k [n \; div \; N_k] \cdot c_{ch,k}[n] \right\} \cdot c_{sh}[n] \\ &= c[n] \cdot c_{sh}[n] \end{aligned} \qquad ()$$

$$c[n] \equiv \sum_{k=1}^{K} G_k b_k [n \; div \; N_k] \cdot c_{ch,k}[n]$$

where *g[t]* is the raised-cosine pulse[1], N_c is the number of samples per chip, and *d[n]* is the composite chip sequence from all users. The received signal is then

$$\begin{aligned} r[t] &= \sum_{q=1}^{L} a_q s[t - \tau_q] \\ &= \sum_{q=1}^{L} a_q \sum_n g[t - \tau_q - nN_c]d[n] \end{aligned} \qquad ()$$

The received signal advanced to the delay of interest is

[1] The chip-matched filter is artificially placed in the transmitter for simplicity of preser

$$
\begin{aligned}
r[nN_c+\tau_p] &= \sum_{q=1}^{L} a_q s[nN_c+\tau_p-\tau_q] \\
&= \sum_{q=1}^{L} a_q \sum_{m} g[nN_c+\tau_p-\tau_q-mN_c]d[m] \\
&= \sum_{q=1}^{L} a_q \sum_{m} g[\tau_p-\tau_q-mN_c]d[m+n]
\end{aligned} \qquad ()
$$

The received signal multiplied by the conjugate of the scrambling codes is

$$
\begin{aligned}
r[nN_c+\tau_p]\cdot c_{sh}^{*}[n] &= \sum_{q=1}^{L} a_q \sum_{m} g[\tau_p-\tau_q-mN_c]c[m+n]c_{sh}[m+n]\cdot c_{sh}^{*}[n] \\
&= \left[\sum_{q=1}^{L} a_q g[\tau_p-\tau_q]\right]\cdot c[n]+w[n] \\
&= \tilde{a}_p\cdot c[n]+w[n]
\end{aligned} \qquad ()
$$

$$
\tilde{a}_p \equiv \left[\sum_{q=1}^{L} a_q g[\tau_p-\tau_q]\right]
$$

This result can now be demultiplexed using the 512 x 512 FHT. Since 512 = 2^9, the FHT proceeds in 9 stages. After the first two stages the SF 4 symbols can be extracted. Similarly, after k stages the SF 2^k symbols can be extracted. The amplitudes $\tilde{a}_p$ can be determined from the embedded pilot symbols, or searcher-receiver estimates can be used. If embedded pilot symbols are used the measurements M_{pk} of the pth multipath of the kth user is in the form

$$
M_{pk} = \tilde{a}_p G_k \qquad ()
$$

which includes the user gain factor. After measurements are taken for all multipaths and all users for a given slot, the multipath amplitudes and user gains can be separated by determining the dominant left and right singular vectors of the rank-1 matrix M_{pk} (aside from an arbitrary scale factor which can be given to either amplitudes or the gains). One the approximate amplitudes $\tilde{a}_p$ are known the actual amplitudes a_p are determined by inverting the diagonally dominant system of equations

$$
\begin{aligned}
\tilde{a}_p &= \sum_{q=1}^{L} a_q g[\tau_p-\tau_q] \\
&= \sum_{q=1}^{L} g_{pq} a_q
\end{aligned} \qquad ()
$$

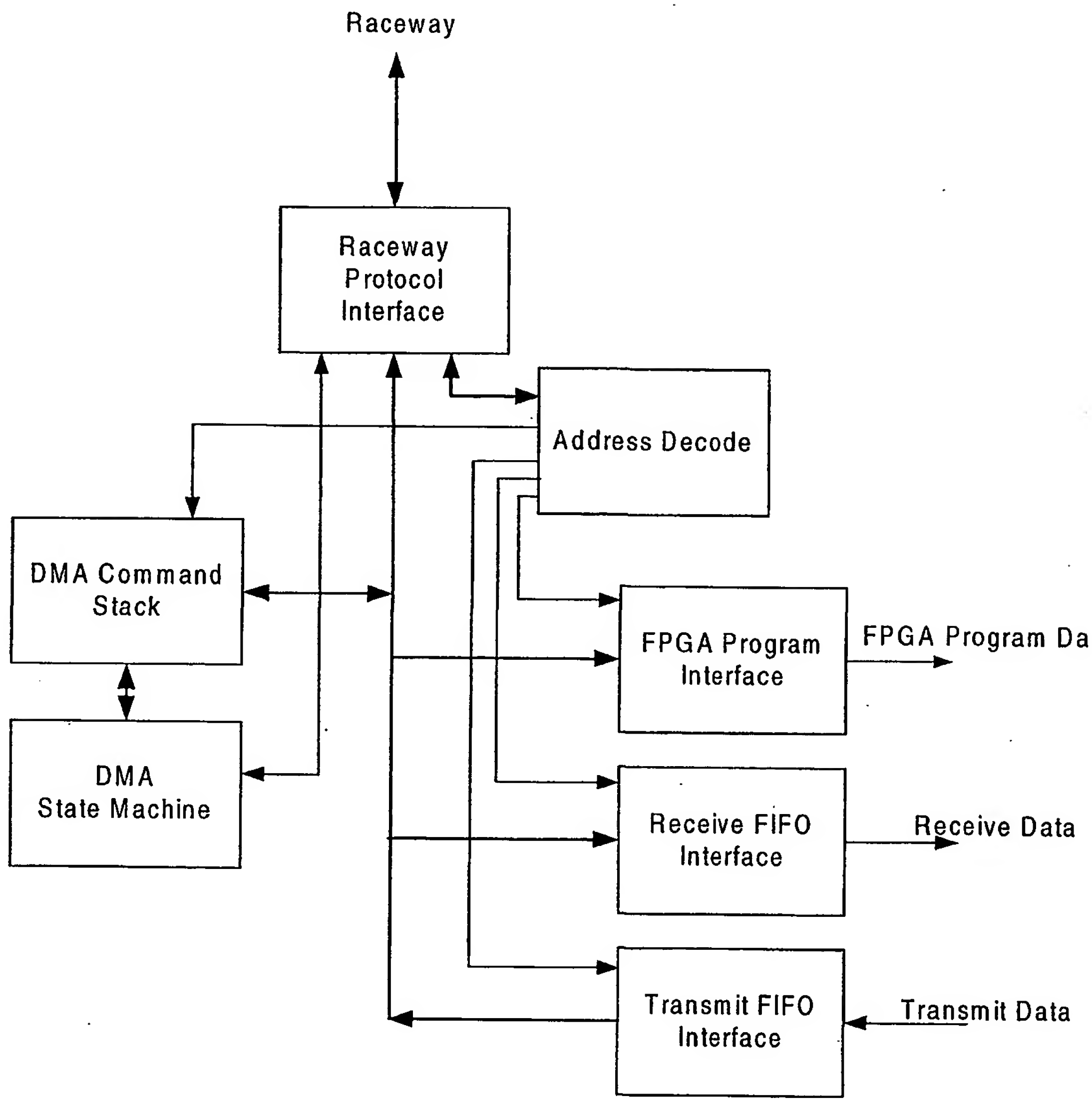
$$
g_{pq} \equiv g[\tau_p-\tau_q]
$$

The chip filter *h[t]* for reconstructing the interference signal is

$$\begin{aligned} r[t] &= \sum_{q=1}^{L} a_q s[t-\tau_q] \\ &= \sum_{q=1}^{L} a_q \sum_{n} g[t-\tau_q - nN_c] d[n] \\ &= \sum_{n} \left[\sum_{q=1}^{L} a_q g[t-\tau_q - nN_c] \right] d[n] \\ &= \sum_{n} h[t-nN_c] \cdot d[n] \end{aligned} \qquad ()$$

$$h[t] \equiv \sum_{q=1}^{L} a_q g[t-\tau_q]$$

Raceway DMA Engine



Possible DSP Raceway Architecture

Type	Memo
Project	MCW-DSP
Current Date	1/31/01
Author(s)	Paul Cantrell
Current Revision	0.1
File	

Revisions

Revision Date	Author	Version	Reason for changes
1/31/01	Paul Cantrell	0.1	Initial Revision

This document contains information which is Proprietary and Confidential to Mercury Computer Systems, Inc. and must not be reproduced or disclosed without Mercury's prior written authorization.

H.A. Bootstrap Functional Design Specification

Table of Contents

H.A. Bootstrap Functional Design Specification

1 Purpose

The purpose of this memo is to document parts of the discussion we have been having on how the TI 6414 DSP may connect to the raceway.

2 Glossary

EMIF – A port on the DSP 6000 series peripheral bus which allows the connection of memory devices.

SDRAM – In the context of this memo, means the main external memory of the TI DSP – the one which contains the program and data.

3 Overview

So far, a proposed architecture is that we use the second EMIF (External Memory Inter-Face) of the TI 6414 DSP to connect to a dual ported RAM. Raceway transfers actually access the RAM, and then additional processing takes place on the DSP to move the data to the correct place in SDRAM. In fact, if the dualport RAM is not large enough to buffer an entire Raceway transfer, then there will have to be a messaging protocol between the two endpoint DSPs wishing to exchange messages (because the message will have to be fragmented in order to not exceed the reserved buffer space).

An additional restriction of this design is that as more Raceway endpoints are added, the size of the dualport RAM needs to be increased, or the maximum fragment size needs to shrink, such that the RAM is big enough to contain at least 2*F*N*P buffers of size F, where F is the size of the fragment, N is the number of Raceway endpoints with which this DSP can exchange messages, P is the number of parallel transfers which can be active on any endpoint at a time, and the constant 2 represents double buffering so that one buffer can be transferred to/from the Raceway, while a second buffer can be transferred to the DSP. The constant becomes 4 if you want to be able to emulate a full duplex connection. With a 4 node system, this might be 4*8K*4*4 or 512K plus a little extra for bookkeeping information. This probably means the minimum size is 1M bytes for the dual port device.

4 Problem Identification

There are several characteristics of this architecture which could prove problematic:

4.1 Requirement For A Fragment/Defragment Protocol

Raceway transfers can currently be very long. This architecture would require a protocol for breaking transfers down into fragments. If the DSP is sourcing a transfer greater than the fragment size, then it has to either dedicate itself for the period of the transfer to programming the DMA engine, or it has to respond to interrupts as each fragment is transferred. In either case, there is a substantial performance impact above and beyond the normal performance hit due to memory bandwidth utilization.

If the DSP is on the receiving end of a Raceway transfer, a similar process has to take place, except that there must be an interrupt to get the attention of the DSP (polling would not be sufficient in such a case).

Beyond the performance hit such a protocol would impose on the DSP, there is a major disadvantage in that only endpoints willing to implement this protocol can exchange data with the DSP. It is in effect, defining a defacto standard subset of Raceway. This is a major interoperability issue (you can no longer plug a board of DSPs into a fabric and have them work as a standard Raceway Adjunct Processor).

4.2 Requirement For The DSP To Be Running Code

If the DSP is involved in the Raceway transfers, then the DSP must already be running in order to perform Raceway transfers. This will require that all nodes on the Raceway be self booting.

4.3 Lower Transfer Rates

Raceway is less efficient with smaller transfer sizes. If the fragment size is kept small to minimize dual port ram requirements, then aggregate Raceway transfer rates will be lower because of less effiicient utilization of the fabric.

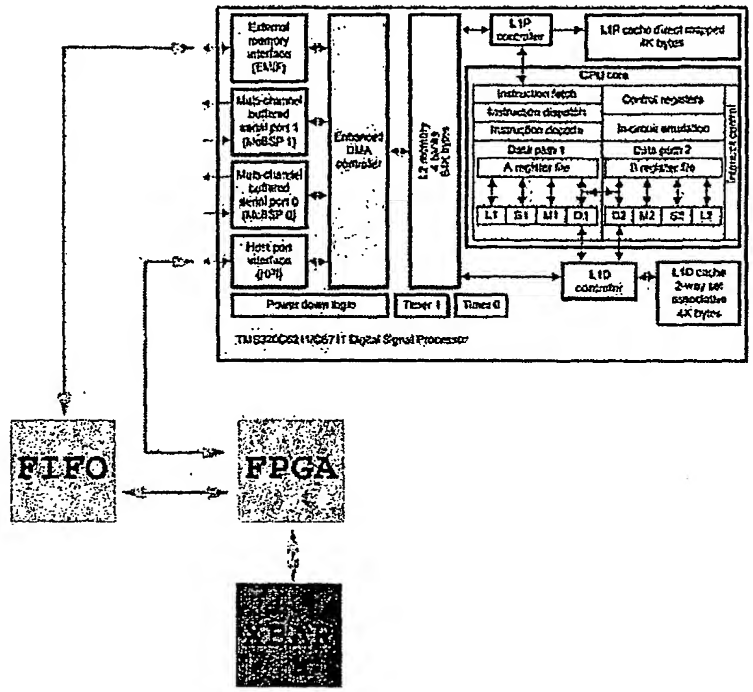
4.4 It Is Different

By changing the way Raceway works, we initiate a significant departure from the way all current Mercury systems work. While there are many other possible architectures which will perform well, it is inherently risky to change a fundemental model of how our multiprocessors communicate.

5 An alternative Architecture

It may be possible to implement a different architecture which addresses some of these shortcomings.

H.A. Bootstrap Functional Design Specification



5.1 Architecture Description

The proposed architecture still has approximately the same hardware as the existing architecture. The changes are in the way that the Raceway transfers move between SDRAM and the Raceway.

In the proposed architecture, the FPGA connects to both the buffering device (dual port RAM or FIFO) and the DSP. The connection to the buffering device (hereafter FIFO) is used to move Raceway data to/from the FIFO.

The second connection is to the DSP Host Port. Dave currently believes this is a moderately high performance interconnect – on the order of 75 Mbytes per second. This interconnect could itself be used to move data to/from the DSP. The host port can access data in the DSP on-chip memory, as well as any of the peripheral devices, including the SDRAM. However, 75Mbytes per second is pretty slow compared to normal Raceway bandwidth, and we think we can do better.

The 6414 contains a second EMIF which can be attached to the FIFO (this is similar to what the current architecture proposal intends). The difference in this proposed architecture is that rather than have the DSP program the DMA engine

to move data between the FIFO and the DSP/SDRAM, we propose that the FPGA can program the DMA engine directly via the Host Port.

The Host Port is a peripheral like the EMIF and the Serial Ports. The difference is that the Host Port can master transfers into the DSP datapaths, i.e. it can read and write any location in the DSP. Because the Host Port can access the DMA Controller (we think), it can be used to initiate transfers via the DMA engine.

The advantage of this architecture is that Raceway transfers can be initiated without the cooperation of the DSP. Thus, the DSP does not have to be self booting. Performance is increased in two ways: the DSP is free to continue to compute while Raceway transfers take place, and performance on the Racway is increased because there is no need to fragment messages.

The internal datapaths of the DSP are flexible enough that we can control which devices have priority access to memory and datapath. Specifically, we can choose to give Raceway transfers priority over the CPU, or vice versa.

5.2 Synchronization Issues

There is an issue to be solved in how we match data rates between Raceway and the DSP. The EMIF looks to the DSP as if it were a memory, thus it is reasonable for the DSP to assume it can get at the data it needs at any time. However, if we indeed use a FIFO to buffer data, the implication is that there is a way to hold off the DSP when we are waiting for the Raceway to empty or fill our FIFO. A possibility is that the buffer device remains a dual port RAM rather than a FIFO, and the FPGA actually does a fragment/defragment into the RAM, and then programs the DMA engine to move that fragment into/out-of the DSP. This starts to look somewhat like the original architecture, except that because the FPGA performs the frag/defrag, the actual transfers over the Raceway can be arbitrarilly sized (assuming we can throttle the Raceway).

Synchronization remains one of the larger problems to be solved with this proposed architecture.

5.3 Sample Transfers

In order to illustrate how this architecture would work, two examples are given. The first example is when the Raceway attempts to read data out of the DSP memory.

5.3.1 Raceway Reading DSP Memory

In this example, we assume that another DSP is trying to read the SDRAM of the local DSP.

1) The FPGA detects a Raceway packet arriving, and decodes that it is a read of address 0x10000 (for instance).

2) The FPGA writes over the Host Port Interface in order to program the DMA engine. It programs the DMA engine to transfer data starting at location 0x10000 (a location in the primary EMIF corresponding to a location in SDRAM) to a location in the secondary EMIF (the buffer

device/FIFO). As data arrives in the buffer device, the FPGA reads the data out of the buffer device, and moves it onto the Raceway. When the proper number of bytes have been moved, the DMA engine finishes the transfer, and the FPGA finishes moving data from the FIFO to the Raceway.

5.3.2 Raceway Writing DSP Memory

In this example, we assume that another DSP is trying to write to the SDRAM of the local DSP.

1) The FPGA detects a Raceway packet arriving, and decodes that it is a write of location 0x20000 (for instance).

2) The FPGA fills some amount of the buffer device with data from the Raceway, and then:

3) The FPGA writes over the Host Port Interface in order to program the DMA engine. It programs the DMA engine to transfer data from the buffer device (secondary EMIF) and to write it to the primary EMIF at address 0x20000.

4) At the end of the transfer, we could either interrupt the DSP to signal that a Raceway packet has arrived, or we can use the standard Mercury method of polling a location in the SMB to see whether the transfer has completed yet.

5.4 Additional Thoughts

1) We need to verify that the Host Port Interface can program the DMA engine. The documentation on the 6201 clearly states that it can write to any location in internal memory, and to anywhere on the peripheral bus, however the DMA engine/controller is the datapath controller for all that, so it is always possible that there is a special case which does not allow writing of the DMA engine/controller registers from HPI. The chance of this being so is quite remote, but needs to be verified.

2) We need to understand the transfer rates and latencies of the HPI. This architecture relies on fairly low latency access through the HPI, otherwise more buffering space would be required, and at some point bandwidth begins to be affected.

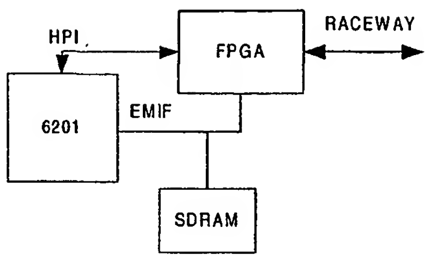
3) We need to understand the limitations of Raceway with respect to throttling, etc. The best case would be that Raceway can provide data as fast as the EMIF can take it (so we wouldn't worry about having data ready when EMIF wanted it), and also for Raceway to be able to be throttled so that it can take the data at the rate the EMIF can provide it. The more the reality deviates from this best case scenerio, the more extra logic is required in the FPGA until at some point complexity may prevent the architecture from being viable.

4) What we currently know about the 6414 is actually educated guesses based on documentation of earlier DSPs. We are making some assumptions about how TI will have enhanced their chip.

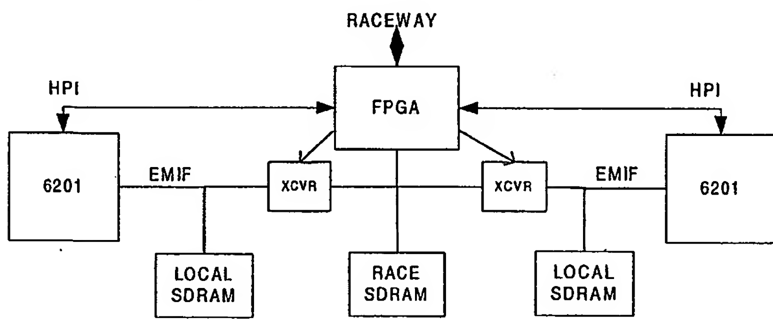
5) If/when TI ever puts a RapidIO interface on their DSPs, it will almost certainly look like a high speed HPI, i.e. it will sit on the peripheral bus, have a separate datapath channel, data coming in will simply flow to the correct addresses, and outgoing data transfers will happen by programming the DMA engine to send data to the RapidIO peripheral address. This proposed architecture looks almost exactly like that, and so probably will not require major changes to use a RapidIO enhanced DSP.

6) There are probably more thoughts.. but this is probably a good start…

6201 Design Options



Option 1



Option 2

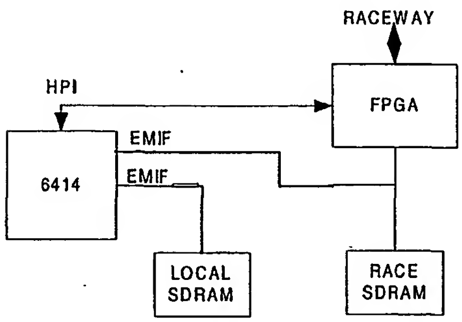
Option 1 is the original proposal submitted at the DSP meeting Monday. Option 2 was created during the meeting.

The main shortfall in Option 1 is the sharing of the EMIF bus between the 6201 and the Raceway DMA FPGA. During DMA operations over the Raceway, the 6201 will not have access to the EMIF interface. Any data or ***instruction*** fetches from SDRAM will stall. Given the relatively small size of the internal SRAM, this will impose a significant penalty to the operation of the 6201. Option 1 also requires the FPGA to take over SDRAM refresh operation when it takes control of the EMIF bus. This passing back and forth of the refresh task will not be clean.

Option 2 places a bi-directional transceiver between the 6201's EMIF bus and the Raceway SDRAM. This allows the 6201 to process data and fetch instructions without any interruption from it's local SDRAM while the DMA FPGA is accessing the Raceway SDRAM. The HPI interface is used by the 6201 to program the DMA engine and by the DMA engine to indicate the DMA complete status to the FPGA. Option 2 also lends itself to a dual 6201 node per raceway interface. Decode logic, controlling access to the Raceway SDRAM can be designed in a number/combination of ways:

Total access to both 6201s
Separate areas for each 6201
Read but no write to the other 6201's memory space
A separate common area accessible to both for message passing
The ability of one 6201 to go through the transceiver to the others local SDRAM (not recommended)

For a migration story to the 6414, Option 2 is a better sell, Option 3 shows the 6414 design, the transceiver is stripped off and the Raceway SDRAM is connected to the second EMIF. The design will go to one DSP per raceway due to the increased in processing power of the 6414.



Option 3

Mercury Computer Systems, Inc.
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Memorandum

To: Jonathan Schonfeld Date: 23-FEB-2001

From: Nmf

Subject: An Efficient WCDMA Receiver Design based on the FFT File Ref: mjv-019-efficient_wcdma_receiver.doc

1. Introduction

Typical processing:

Signal is sampled at N samples per chip.
Despread by
upsampling chipping sequence by interpolating and using the RRC chip pulse matched filter as an interpolation filter
Multiplying digitized receive signal by upsampled and interpolated chip sequence
Accumulate (integrate) results for an entire DPCCH symbol.
Repeat at the early lead and late lag sample offset values to calculate delay locked loop variables
Sweep the code correlator N*256 lags to determine code synchronization and channel response

Spreading sequence is 256 chips long
Typical filter is 12 chips long
typical oversampling rate on the receiver is N=8

Key calculations
Interpolation of the spreading code – precomputed and stored

Correlation process: N*256 CMAC
Correlation repeated for N*256 + 2 (DLL) times
Total CMACS: N*256 * (N * 256 + 2) = N^2*65536 + 512 * N
For N = 8, this results in: 4,198,400 CMAC
1 CMAC = 4 RMUL + 2 RADD = 6 ROP

Results in 25,190,400 Real operations
At 15000 Hz symbol rate, need: 378 GOP/s

2. A New Design

Use of FFT to perform efficient circular convolution of spreading code sequence

Results in
Short code synchronization (chip sync only, not slot or frame)
DPCCH demodulation
Early and late Delay Locked Loop variables
Rough channel estimate values for an entire symbol worth of differential delay

Polyphase signal processing

Digitize the signal at an Nx oversample rate and filter with the RRC filter and split into N streams at the 1x rate.

Compute the complex conjugate of the FT of the spreading code sequence at the chip rate – precomputed and stored

Computation:
Filter data at Nx oversample rate and split into N streams at 1x rate

For each stream,
    Compute 256 point FFT
    Complex multiply FFT with stored FFT values of spreading code
    Inverse 256-point FFT

Ops calculation:
Input filter: could be done using FFT as well.
but for time domain processing: 8*256 points, filter length 96 =>
96 RMUL per point, 95 RADD per point,
Total of 19608 RMUL, 194,560 RADD per symbol ==> 391,168 ROP per symbol
I and Q streams, => 782336 ROP

Stream processing (8 streams)
    Radix 4 FFT: 256*4*(4 CMUL + 8 CADD) = 34,816 ROP
    256 CMUL = 1536 ROP
    Radix 4 IFFT: 256*4*(4 CMUL + 8 CADD) = 34,816 ROP
    TOTAL per stream: 71168 ROP

Total stream calcs: 569,344 ROP

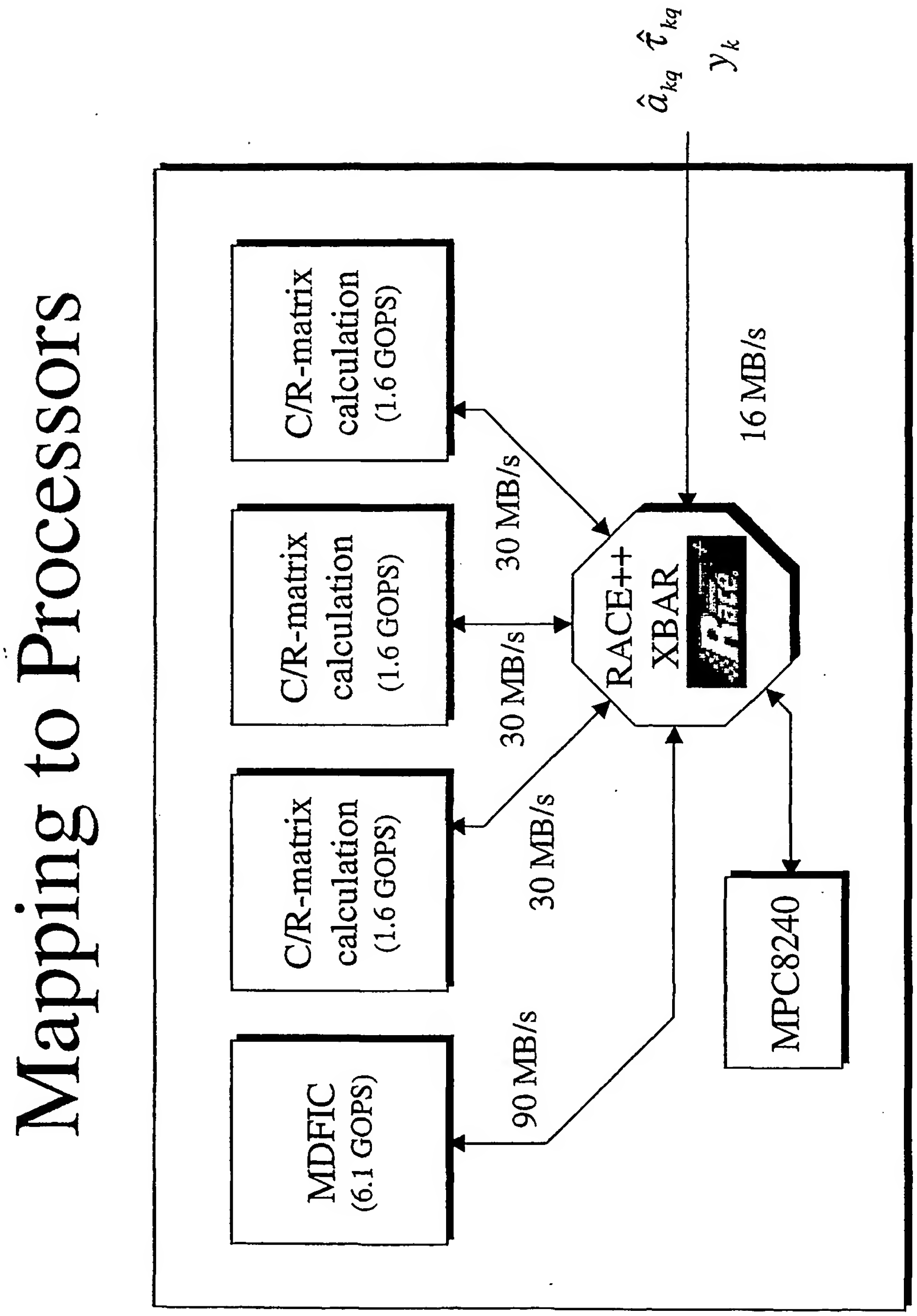
Total ops per second at 15000 Hz symbol rate is: 20.3 GOPS
more than 18 times more efficient than traditional approach.

Also, the DLL circuitry can be eliminated since the entire channel response is calculated at the symbol rate.

FFT numbers may be off by a factor of 2 larger in the number of complex multiplications needed.

3. References:

Scholtz, et. al. Spread Spectrum Handbook.
Proakis and Manolakis. Introduction to Digital Signal Processing. Macmillian, 1988.


Mapping to Processors
$\hat{a}_{kq}$ $\hat{\tau}_{kq}$
y_k
16 MB/s
C/R-matrix calculation (1.6 GOPS)
C/R-matrix calculation (1.6 GOPS)
C/R-matrix calculation (1.6 GOPS)
MDFIC (6.1 GOPS)
30 MB/s
30 MB/s
30 MB/s
90 MB/s
RACE++ XBAR
MPC8240

Practical Implementation of an Iterative Hard-Decision MUD Algorithm for the UMTS FDD Uplink

John H. Oates
Mercury Computer Systems, Inc.
Wireless Communications Group
199 Riverneck Road
Chelmsford, MA 01824-2820 USA
Tel: 978-256-0052 x 1659
FAX: 978-256-8596
E-mail: joates@mc.com
Technical Area: 03

Introduction

Multi-User Detection (MUD) has been shown to provide a number of significant benefits[1][2]. These include increased system capacity, increased range, enhanced Quality of Service (QoS), improved near-far resistance, extended battery life, and reduced handset transmit power. This paper describes the practical implementation of Multi-User Detection (MUD) for the UMTS uplink using short codes. The focus is on practical implementation details such as efficient implementation of the calculations, processing requirements, latencies, MUD efficiency, and mapping to hardware.

The use of short codes allows MUD to be performed at the symbol rate. As such MUD can be introduced into a conventional Base-Transceiver-Station (BTS) as an enhancement to the Matched-Filter (MF) RAKE receiver. The MUD processing takes the MF detection statistics, performs interference cancellation, and then delivers improved hard or soft-decision symbol estimates to the symbol-rate BTS processing functions. The MUD processing introduces only a few milliseconds latency. Because of the reduced computational complexity of MUD operating at the symbol rate the entire MUD functionality can be implemented in software on a single card or daughter card populated with a minimal number of processors. We present here an implementation of an iterative hard-decision Interference Cancellation (IC) algorithm on four Power PC 7410 processors. The processors are connected together with a high-bandwidth RACE++ interconnect fabric.

In order to perform MUD at the symbol rate the correlation between the user channel-corrupted signature waveforms must be calculated. These correlations are stored as elements of matrices, here referred to as the R-matrices. Since the channel is continually changing these correlations must be updated in real time. There are two elements to updating the R-matrices. The first part is based on the user code correlations. These depend on the relative lag between the various user multipath components. It is assumed that these lags change with a time constant of about 400 ms. The second part is due to the fast variation of the Rayleigh-fading multipath amplitudes. It is assumed that these amplitudes are changing with a time constant of about 1.33 ms. The R-matrices are used to cancel the multiple access interference through the Multi-stage Decision-Feedback Interference Cancellation (MDFIC) technique.

UMTS Uplink Multi-rate Signal Model and RAKE Processing

We derive here the equations describing the MF outputs based on the WCDMA transmitted waveform. The users accessing the system will hereafter be referred to as *physical users*. Each physical user is regarded as a composition of *virtual users*. Each virtual user transmits a single bit per symbol period, where by *symbol period* we mean a time duration of 256 chips (i.e. 1/15 ms). The number of virtual users, then, for a given physical user is equal to the number of bits transmitted in a symbol period. At a minimum each active physical user is composed of two virtual users, one for the Dedicated Physical Control Channel

(DPCCH)[3] and one for the Dedicated Physical Data CHannel (DPDCH). If the physical user is a data user with Spreading Factor (SF) less than 256 then there are $J = 256/SF$ data bits and one control bit transmitted per symbol period. Hence for the rth physical user with data-channel spreading factor SF_r, there are a total of $1 + 256/SF_r$ virtual users. The total number of virtual users is denoted

$$K_v \equiv \sum_{r=1}^{K}\left[1+\frac{256}{SF_r}\right] \tag{1}$$

The transmitted waveform for the rth physical user can be written as

$$\begin{aligned} x_r[t] &= \sum_{k=1}^{1+J_r}\beta_k\sum_m s_k[t-mT]b_k[m] \\ s_k[t] &\equiv \sum_{p=0}^{N-1}h[t-pN_c]c_k[p] \end{aligned} \tag{2}$$

where t is the integer time sample index, $T = NN_c$ is the data bit duration, $N = 256$ is the short-code length, N_c is the number of samples per chip, and where $\beta_k = \beta_c$ if the kth virtual user is a control channel and $\beta_k = \beta_d$ if the kth virtual user is a data channel. The multipliers β_c and β_d are constants used to select the relative amplitudes of the control and data channels. At least one of these constants must be equal to 1 for any given symbol period m. The waveform $s_k[t]$ is referred to as the transmitted signature waveform for the kth virtual user. This waveform is generated by passing the spread code sequence $c_k[n]$ through a root-raised-cosine pulse shaping filter $h[t]$. If the kth virtual user corresponds to a data user with spreading factor less than 256 then the code $c_k[n]$ still has length 256, but only N_k of the 256 elements are non-zero, where N_k is the spreading factor for the kth virtual user. The non-zero values are extracted from the code $C_{ch,256,64}$ $S_{sh}[n]$[3]. The W-CDMA standard actually allows for up to six DPDCHs to be multiplexed with a single DPCCH. This functionality is not presently incorporated in the MUD algorithms described below.

The baseband received signal can be written

$$\begin{aligned} r[t] &= \sum_{k=1}^{K_v}\sum_m \tilde{s}_k[t-mT]b_k[m] + w[t] \\ s_k[t] &\equiv \sum_{q'=1}^{L}a_{kq'}s_k[t-\tau_{kq'}] \end{aligned} \tag{3}$$

where $w[t]$ is receiver noise, $\tilde{s}_k[t]$ is the channel-corrupted signature waveform for virtual user k, L is the number of multipath components, and $a_{kq'}$ are the complex multipath amplitudes. The amplitude ratios β_k are incorporated into the amplitudes $a_{kq'}$. Notice that if k and l are two virtual users corresponding to the same physical user then, aside from scaling the by β_k and β_l, $a_{kq'}$ and $a_{lq'}$, are equal. This is due to the fact that the signal waveforms of all virtual users corresponding to the same physical user pass through the same channel. The waveform $s_k[t]$ is now the received signature waveform for the kth virtual user. This waveform is identical to the transmitted signature waveform given in Equation (2) except that the root-raised-cosine pulse $h[t]$ is replaced with the raised-cosine pulse $g[t]$.

Thus far the received signal has been match-filtered to the chip pulse. It must next be match-filtered by the user code-sequence filter. The resulting detection statistic is denoted here as y_k, the matched-filter output for the kth virtual user. Since there are K_v codes, there are K_v such detection statistics, which are collected into a column vector $y[m]$ for the mth symbol period. The matched-filter output $y_l[m]$, for the lth virtual user can be written

$$y_l[m] \equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}_{lq}^{*}\cdot\frac{1}{2N_l}\sum_n r[nN_c+\hat{\tau}_{lq}+mT]\cdot c_l^{*}[n]\right\} \tag{4}$$

where $\hat{a}^*_{lq}$ is the estimate of a^*_{lq}, $\hat{\tau}_{lq}$ is the estimate of τ_{lq}, N_l is the (non-zero) length of code $c_l[n]$, and $\eta_l[m]$ is the match-filtered receiver noise. Substituting $r[t]$ from Equation (3) above gives

$$
\begin{aligned}
y_l[m] &\equiv \sum_{m'}\sum_{k=1}^{K_v} \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}^*_{lq}\cdot\frac{1}{2N_l}\sum_n \tilde{s}_k[nN_c+\hat{\tau}_{lq}+m'T]\cdot c^*_l[n]\right\} b_k[m-m']+\eta_l[m] \\
&= \sum_{m'}\sum_{k=1}^{K_v} r_{lk}[m'] b_k[m-m']+\eta_l[m] \\
r_{lk}[m'] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L}\hat{a}^*_{lq}\cdot\frac{1}{2N_l}\sum_n \tilde{s}_k[nN_c+\hat{\tau}_{lq}+m'T]\cdot c^*_l[n]\right\} \\
&= \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}^*_{lq}a_{kq'}\cdot\frac{1}{2N_l}\sum_n s_k[nN_c+m'T+\hat{\tau}_{lq}-\tau_{kq'}]\cdot c^*_l[n]\right\} \\
&= \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}^*_{lq}a_{kq'}\cdot\frac{1}{2N_l}\sum_n\sum_p g[(n-p)N_c+m'T+\hat{\tau}_{lq}-\tau_{kq'}]c_k[p]\cdot c^*_l[n]\right\}
\end{aligned} \tag{5}
$$

The terms for $m'\neq 0$ result from asynchronous users.

MUD Algorithm and Functions

A vast number of MUD algorithms have been proposed [1][2]. Many of these are too computationally complex to be implemented with current technology. The linear-iterative class of MUD algorithms [4][5][6] are the least computationally complex. For this class of algorithms software implementation is feasible. The hard-decision variants of these algorithms also enjoy a significant performance advantage in that they do not tend to amplify other-cell interference. The down side is that performance degrades under high input BER. Since channel decoding reduces the BER by orders of magnitude, it is possible to be operating with raw channel BERs as high as 10%. A number of methods have been proposed to address this issue, including the null-zone detector [4], and partial interference cancellation [4][5][6]. We employ partial interference cancellation in conjunction with a new thresholding technique which reduces computational complexity. Our method provides excellent performance under high input BER.

The implementation of MUD at the symbol rate can be divided into two functions. The first function is the calculation of the R-matrix elements. The second function is interference cancellation, which relies on knowledge of the R-matrix elements. The calculation of these elements and the computational complexity are described in the following section. Computational complexity is expressed in Giga Operations Per Second (GOPS). The subsequent section describes the MUD IC function. The method of interference cancellation employed is Multistage Decision Feedback IC (MDFIC)[2][7].

R-matrix

From Equation (5) above, the R-matrix calculations can be divided into three separate calculations, each with an associated time constant for real-time operation, as follows

$$
\begin{aligned}
r_{lk}[m'] &= \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{a_{lq}^{*}a_{kq'}\cdot\frac{1}{2N_I}\sum_{n}\sum_{p}g[(n-p)N_c+m'T+\tau_{lq}-\tau_{kq'}]c_k[p]\cdot c_l^{*}[n]\right\} \\
&= \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{a_{lq}^{*}a_{kq'}\cdot C_{lkqq'}[m']\right\}
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
C_{lkqq'}[m'] &\equiv \frac{1}{2N_I}\sum_{n}\sum_{p}g[(n-p)N_c+m'T+\tau_{lq}-\tau_{kq'}]c_k[p]\cdot c_l^{*}[n] \\
&= \frac{1}{2N_I}\sum_{m}g[mN_c+m'T+\tau_{lq}-\tau_{kq'}]\sum_{n}c_k[n-m]\cdot c_l^{*}[n] \\
&= \frac{1}{2N_I}\sum_{m}g[mN_c+m'T+\tau_{lq}-\tau_{kq'}]\Gamma_{lk}[m]
\end{aligned}
$$

$$
\Gamma_{lk}[m] \equiv \sum_{n}c_k[n-m]\cdot c_l^{*}[n]
$$

where we have omitted the hats indicating parameter estimates. Hence we must calculate the R-matrices, which depend on the C-matrices ($C_{lkqq'}[m']$), which depend on the Γ-matrix ($\Gamma_{lk}[m]$). The Γ-matrix has the slowest time constant. This matrix represents the user code correlations for all values of offset m. For the case of 100 voice users the total memory requirement is 21 MB based on two bytes (real and imaginary parts) per element. This matrix is updated only when new codes (new users) are added to the system. Hence this is essentially a static matrix. The computational requirements are negligible. The most efficient method of calculation depends on the non-zero length of the codes. For high data-rate users the non-zero length of the codes is only 4 chips long. For these codes a direct convolution is the most efficient method to calculation the elements. For low data-rate users it is more efficient to calculation the elements using the FFT to perform the convolutions in the frequency domain.

The C-matrix is calculated from the Γ-matrix. These elements must be calculated whenever a users delay lag changes. For now assume that on average each multipath component changes every 400 ms. The length of the $g[]$ function is 48 samples. Since we are oversampling by 4, there are 12 multiply-accumulations (real x complex) to be performed per element, or 48 operations per element. When there are 100 low-rate users on the system (200 virtual users) and a single multipath lag (of 4) changes for one user a total of $(1.5)(2)K_vLN_v$ elements must be calculated. The factor of 1.5 comes from the 3 C-matrices ($m' = -1, 0, 1$), reduced by a factor of 2 due to a conjugate symmetry condition. The factor of 2 results because both rows and columns must be updated. The factor N_v is the number of virtual users per physical user, which for the lowest rate users is $N_v = 2$. In total then this amounts to 230400 operations per multipath component per physical user. Assuming 100 physical users with 4 multipath components per user, each changing once per 400 ms gives 230 MOPS.

The R-matrices are calculated from the C-matrices. From Equation (6) above the R-matrix elements are

$$
r_{lk}[m'] = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{*}a_{kq'}\cdot C_{lkqq'}[m']\right\} = \mathrm{Re}\left\{a_l^{H}\cdot C_{lk}[m']\cdot a_k\right\}
\tag{7}
$$

where a_k are $L \times 1$ vectors, and $C_{lk}[m']$ are $L \times L$ matrices. The rate at which these calculations must be performed depends on the velocity of the users. The selected update rate is 1.33 ms. If the update rate is too slow such that the estimated R-matrix values deviate significantly from the actual R-matrix values then there is a degradation in the MUD efficiency. Figure 1 below shows the degradation in MUD efficiency versus user velocity for an update rate of 1.33 ms, which corresponds to two WCDMA time slots. The plot indicates that there is high MUD efficiency for users with velocity less than about 100 km/hr. The plot indicates that the interference corresponding to fast users is not cancelled as effectively as the interference due to slow users. For a system with a mix of fast and slow users the resulting MUD efficiency is a average of the MUD efficiency for the various user velocities.

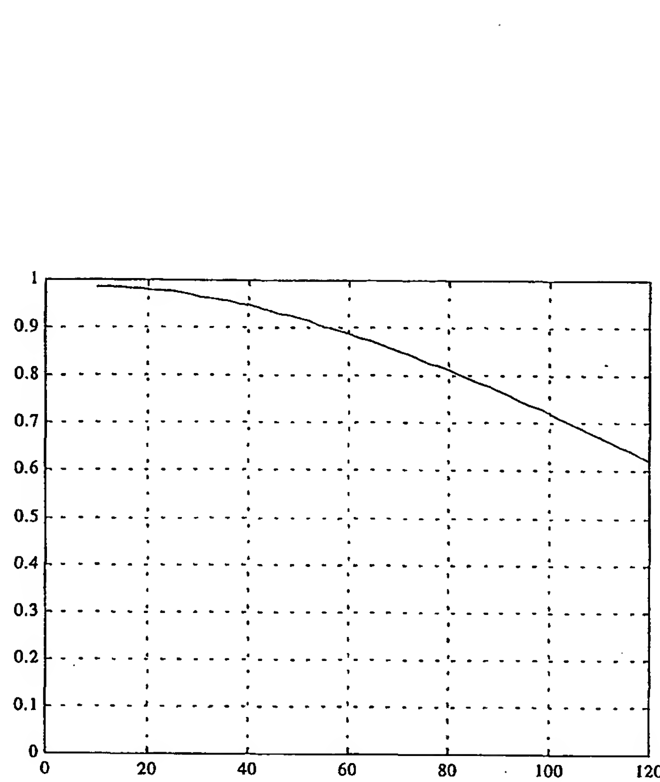


Figure 1. MUD efficiency versus user velocity in km/hr

From Equation (7) the calculation of the R-matrix elements can be calculated in terms of an X-matrix which represents amplitude-amplitude multiplies

$$
\begin{aligned}
r_{lk}[m'] &= \mathrm{Re}\{tr[a_l^H \cdot C_{lk}[m'] \cdot a_k]\} = \mathrm{Re}\{tr[C_{lk}[m'] \cdot a_k \cdot a_l^H]\} \equiv \mathrm{Re}\{tr[C_{lk}[m'] \cdot X_{lk}]\} \\
&= tr[C_{lk}^R[m'] \cdot X_{lk}^R] - tr[C_{lk}^I[m'] \cdot X_{lk}^I]
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
X_{lk} &\equiv a_k \cdot a_l^H \equiv X_{lk}^R + jX_{lk}^I \\
C_{lk}[m'] &\equiv C_{lk}^R[m'] + jC_{lk}^I[m']
\end{aligned}
$$

The advantage of this approach is that the X-matrix multiplies can be reused for all virtual users associated with a physical user and for all m' (i.e. $m' = 0, 1$). Hence these calculations are negligible when amortized. The remaining calculations can be expressed as a single real dot product of length $2L^2 = 32$. The calculations are be performed in 16-bit fixed-point math. The total operations is thus $1.5(4)(K_v L)^2 = 3.84$ Mops. The processing requirement is then 2.90 GOPS. The X-matrix multiplies when amortized amount to an additional 0.7 GOPS. The total processing requirement is then 3.60 GOPS.

MDFIC

From Equation (5) above the matched-filter outputs are given by

$$
y_l[m] = r_{ll}[0]b_l[m] + \sum_{k=1}^{K_v} r_{lk}[-1]b_k[m+1] + \sum_{k=1}^{K_v} [r_{lk}[0] - r_{ll}[0]\delta_{lk}]b_k[m] + \sum_{k=1}^{K_v} r_{lk}[1]b_k[m-1] + \eta_l[m] \tag{9}
$$

The first term represents the signal of interest. All the remaining terms represent Multiple Access Interference (MAI) and noise. The MDFIC algorithm iteratively solves for the symbol estimates $\hat{b}_l[m]$ using

$$\hat{b}_l[m] = sign\left\{ y_l[m] - \sum_{k=1}^{K_v} r_{lk}[-1]\hat{b}_k[m+1] - \sum_{k=1}^{K_v} [r_{lk}[0] - r_{ll}[0]\delta_{lk}]\hat{b}_k[m] - \sum_{k=1}^{K_v} r_{lk}[1]\hat{b}_k[m-1] \right\} \tag{10}$$

with initial estimates given by hard decisions on the matched-filter detection statistics, $\hat{b}_l[m] = sign\{y_l[m]\}$. The MDFIC [7] technique is closely related to the SIC and PIC technique. Notice that new estimates $\hat{b}_l[m]$ are immediately introduced back into the interference cancellation as they are calculated. Hence at any given cancellation step the best available symbol estimates are used. This idea is analogous to the Gauss-Siedel method for solving diagonally dominant linear systems.

The above iteration is performed on a block of 20 symbols, for all users. The 20-symbol block size represents two WCDMA time slots. The R-matrices are assumed to be constant over this period. Performance is improved under high input BER if the *sign* detector in Equation (10) is replaced by the hyperbolic tangent detector [6]. This detector has a single slope parameter which is variable from iteration to iteration.

The three R-matrices (R[-1], R[0] and R[1]) are each K_v x K_v in size. The total number of operation then is $6K_v^2$ per iteration. The computational complexity of the MDFIC algorithm depends on the total number of virtual users, which depends on the mix of users at the various spreading factors. For $K_v = 200$ users (e.g. 100 low-rate users) this amounts to 240,000 operations. In the current implementation two iterations are used, requiring a total of 480,000 operation. For real-time operation these operations must be performed in 1/15 ms. The total processing requirement is then 7.2 GOPS. Computational complexity is markedly reduced if a threshold parameter is set such that IC is performed only for values $|y_l[m]|$ below the threshold. The idea is that if $|y_l[m]|$ is large there is little doubt as to the sign of $b_l[m]$, and IC need not be performed. The value of the threshold parameter is variable from stage to stage.

Mapping to Hardware

The above calculations are performed on a single 9"x6" card populated with four Power PC 7410 processors. These processors employ the AltiVec SIMD vector arithmetic-logic unit, which has 32 128-bit vector registers. These registers can hold either 4 32-bit floats, 4 32 bit ints, 8 16-bit shorts, or 16 8-bit chars. Two vector SIMD operation (multiply and accumulate) can be performed by clock. The clock rate used for the current implementation is 400 MHz. The processors, however, can be operated at 500 MHz with higher clock speeds in the near future. Each processor has 32KB of L1 cache and 2MB of 266MHz L2 cache. The maximum theoretical performance of these processors is thus 3.2 GFLOPS, 6.4 GOPS (16-bit), or 12.8 GOPS (8-bit). The current implementation used a combination of floating-point, 16-bit fixed-point and 8-bit fixed-point calculations.

The four PPC7410 processors are interconnected with a RACE++ 266MB/s 8-port switched fabric as shown in Figure 2. The high bandwidth fabric allows transfer of large amounts of data with very low latency so as to achieve efficient parallelism of the four processors. The maximum theoretical performance of the card is thus 51.2 GOPS.

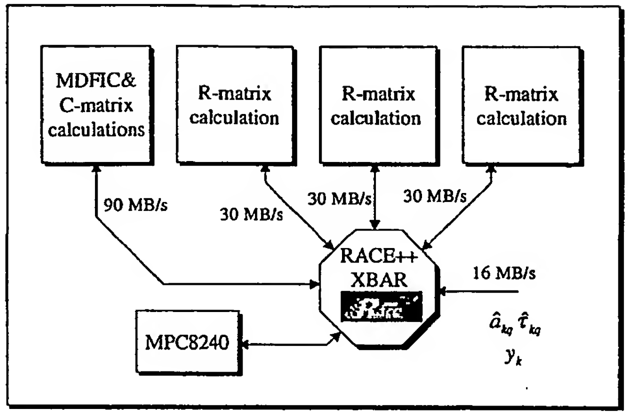


Figure 2. Partitioning of MUD functions across four processors

As shown in Figure 2 the MDFIC and C-matrix calculations are allocated to a single processor. The other three processors are given to the R-matrix calculations which are considerably more complex.

MUD BER Performance

A sample of the Bit Error Rate (BER) performance of the MUD algorithm is shown in Figure 3. For comparison the matched-filter BER is also shown. The figure shows that MUD doubles system capacity.

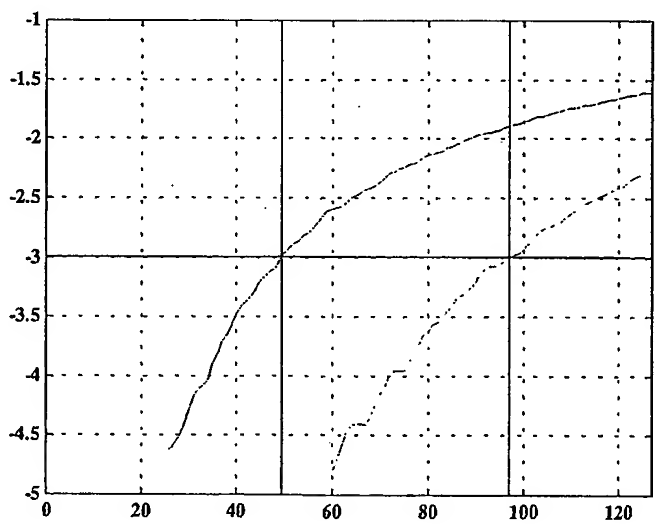


Figure 3. Log10 bit error rate versus system capacity for matched filter (blue) and multiuser detection (red)

The above performance is based on the following assumptions:

- A single receive antenna is used
- The target BER is 0.001

- The percentage of systems users in handoff is 30%
- Other-cell interference is 35% of intra-cell interference. This is lower than the typical value (0.60) used. The reason is that the other-cell users in handoff with the cell of interest are included in the intra-cell interference. This is because the cell of interest is processing these users and hence can cancell there interference using MUD.
- A 4-tap multipath channel is used. Each tap is Rayleigh fading. The composite power of all paths is perfectly power controlled.
- The channel amplitude estimation error is 10%
- The channel delay estimation is ¼ chip
- The activity factor for voice is 0.40
- The relative amplitude of the control channel is $\beta_c = 0.5333$

Conclusions

The current state of processor technology is such that iterative hard-decision MUD for the UMTS uplink can be implemented in software on a single card or daughter card populated with four Power PC 7410 processors, connected together with a high-bandwidth RACE++ interconnect fabric. The use of short codes allows MUD to be performed at the symbol rate. The advantage of symbol-rate processing is that MUD can be introduced into a BTS as an enhancement to the conventional RAKE receiver. The MUD processing takes the MF detection statistics, performs interference cancellation, and then delivers improved hard or soft-decision symbol estimates to the symbol-rate BTS processing functions. The latency introduced is only a few milliseconds. In order to perform MUD at the symbol rate the R-matrices must be updated in real time. There is a minimal degradation in MUD efficiency if these elements are updated at a rate of once per 1.33 ms. The R-matrices are used to cancel the multiple access interference through the MDFIC interference cancellation technique. At a BER of 0.001 the use of the above MUD technique doubles system capacity.

References

[1] A. Duel-Hallen, J. Holtzman, and Z. Zvonar. Multiuser detection for CDMA systems. IEEE Personal Communications, 2(2):46-58, April 1995.

[2] S. Moshavi. Multi-user detection for DS-CDMA communications. IEEE Communications Magazine, pages 124-136, October 1996.

[3] 3G TS 25.213: "Spreading and modulation (FDD)"; 3GPP

[4] D. Divsalar and M.K. Simon. Improved CDMA performance using parallel interference cancellation. IEEE MILCOM, pages pp. 911-917, October 1994.

[5] D. Divsalar, M. Simon, and D. Raphaeli. A new approach to parallel interference cancellation for CDMA. IEEE Global Telecommunications Conference, pages 1452-1457, 1996.

[6] D. Divsalar, M.K. Simon, and D. Raphaeli. Improved parallel interference cancellation for CDMA. IEEE Trans. Commun., 46(2):258-268, February 1998.

[7] T.R. Giallorenzi and S.G. Wilson. Decision feedback multiuser receivers for asynchronous CDMA systems. IEEE Global Telecommunications Conference, pages 1677-1682, June 1993.

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: Long-Code MUD Date: November 3, 2000

1. Introduction

This report briefly describes long-code Multi-User Detection (MUD). Section 2 describes the long-code signal model, which is different from the short-code model. Section 3 describes the matched-filtering operation for long codes and gives a lower bound on the GOPS required for long-code symbol-rate MUD. The lower bound is 19.7 TOPS (i.e. Tera Operations Per Second; 1 TOPS = 1000 GOPS). Because of the extreme computational complexity of symbol-rate MUD for long codes regenerative MUD is examined. It is shown in Section 4 that although regenerative MUD operates at the chip rate, the overall complexity is lower for long codes. Two methods are examined. The first method is a somewhat straight-forward implementation of regenerative MUD. The required computational complexity is shown to be 774.6 GOPS for 100 users. The second method is based on combining impluse trains and subsequently raised-cosine filtering the composite signal. The total computational complexity is shown to be 109.6 GOPS for 100 users. Regenerative MUD is linear in the number of users, so that if the number of users is reduced to 64 the complexity drops to 70.1 GOPS. The complexity is also linear in the number of multipaths subtracted, so that if the number of multipaths subtracted is reduced from 4 to 2 the complexity drops to 35.1 GOPS. It may be desirable for MUD performance to subtract only the two largest multipaths due channel amplitude estimation errors. The above complexity figures are for a single interference cancellation stage. For two stages the computation is doubled. To perform regenerative MUD the baseband antenna stream data must be brought onto the MUD board. The required bandwidth is 123 MB/s. Note that the figures given above can perhaps be reduced through a clever implementation. A block diagram of regenerative MUD is shown to facilitate an investigation into the feasibility of an FPGA or ASIC implementation.

2. Signal Model

The received signal model for short-code WCDMA is given in [1]. When long codes are used the signal model is different since effectively the codes change from symbol to symbol. We present here the WCDMA signal model for long codes. The baseband received signal can be written

$$r[t]=\sum_{k=1}^{2K}\sum_{m}\tilde{s}_{km}[t-mT_k]b_k[m]+w[t] \tag{1}$$

where t is the integer time sample index, $T_k = N_k N_c$ is the data bit duration, which depends on the user spreading factor, N_k is the spreading factor for the kth virtual user, N_c is the number of samples per chip, K is the total number of physical users, $w[t]$ is receiver noise, and where $\tilde{s}_{km}[t]$ is the channel-corrupted signature waveform for the kth virtual user over the mth symbol period. The concept of virtual users is used to account for both the DPDCH and the DPCCH. Hence if there are K physical users, then there are $K_v = 2K$ virtual users. The user signature waveform and hence the channel-corrupted signature waveform vary from symbol period to symbol period since long codes by definition extend over many symbol periods. For L multipath components the channel-corrupted signature waveform for virtual user k is modeled as

$$\tilde{s}_{km}[t]=\sum_{p=1}^{L}a_{kp}s_{km}[t-\tau_{kp}] \tag{2}$$

where a_{kp} are the complex multipath amplitudes. The amplitude ratios β_k are incorporated into the amplitudes a_{kp}. Notice that if k and l are virtual users corresponding to the DPCCH and the DPDCH of the same physical user then, aside from scaling the by β_k and β_l, a_{kp} and a_{lp}, are equal. This is due to the fact that the sig nal waveforms for both the DPCCH and the DPDCH pass through the same channel.

The waveform $s_{km}[t]$ is referred to as the signature waveform for the *kth* virtual user over the mth symbol period. This waveform is generated by passing the spreading code sequence $c_{km}[n]$ through a pulse-shaping filter $g[t]$

$$\begin{aligned} s_{km}[t] &= \sum_{r=0}^{N_k-1} g[t-rN_c]c_{km}[r] \\ &= \sum_{r=0}^{N_k-1} g[t-rN_c]c_k[r+mN_k] \end{aligned} \tag{3}$$

where $g[t]$ is the raised-cosine pulse shape. Since $g[t]$ is a raised-cosine pulse as opposed to a root-raised-cosine pulse, the received signal $r[t]$ represents the baseband signal after filtering by the matched chip filter.

3. Matched filter

The received signal above, which has been match-filtered to the chip pulse, must next be match-filtered by the user code-sequence filter. The resulting detection statistic is

denoted here as $y_k[m]$, the matched-filter output for the kth virtual user over the mth symbol period. Since there are K_v codes, there are K_v such detection statistics, which are collected into a column vector $y[m]$. The matched-filter output $y_l[m]$, for the lth virtual user can be written

$$y_l[m] \equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} r[nN_c + \hat{\tau}_{lq} + mT_l] \cdot c_{lm}^{*}[n]\right\} \tag{4}$$

where $\hat{a}_{lq}^{H}$ is the estimate of a_{lq}^{H}, $\hat{\tau}_{lq}$ is the estimate of τ_{lq}, and $\eta_l[m]$ is the match-filtered receiver noise. Substituting $r[t]$ from Equation (1) above gives

$$\begin{aligned}
y_l[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \left[\sum_{k=1}^{2K} \sum_{m'} \sum_{p=1}^{L} a_{kp} s_{km}[nN_c + \tau_{lkqp}[m,m']] b_k[m'] \right.\right. \\
&\qquad \left.\left. + w[nN_c + \hat{\tau}_{lq} + mT_l] \right] c_{lm}^{*}[n]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} \left[\sum_{k=1}^{2K} \sum_{m'} \sum_{p=1}^{L} a_{kp} s_{km}[nN_c + \tau_{lkqp}[m,m']] b_k[m']\right] \cdot c_{lm}^{*}[n]\right\} + \eta_l[m] \\
&= \sum_{k=1}^{2K} \mathrm{Re}\left\{\sum_{q=1}^{L} \sum_{p=1}^{L} \hat{a}_{lq}^{H} a_{kp} \cdot \sum_{m'} \left[\frac{1}{2N_l} \sum_{n=0}^{N_l-1} s_{km}[nN_c + \tau_{lkqp}[m,m']] \cdot c_{lm}^{*}[n]\right] \cdot b_k[m']\right\} + \eta_l[m] \\
&= \sum_{m'} \sum_{k=1}^{2K} \mathrm{Re}\left\{\sum_{q=1}^{L} \sum_{p=1}^{L} \hat{a}_{lq}^{H} a_{kp} \cdot C_{lkqp}[m,m']\right\} \cdot b_k[m'] + \eta_l[m]
\end{aligned}$$

$$\begin{aligned}
C_{lkqp}[m,m'] &\equiv \frac{1}{2N_l} \sum_{n=0}^{N_l-1} s_{km}[nN_c + \tau_{lkqp}[m,m']] \cdot c_{lm}^{*}[n] \\
\eta_l[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^{H} \cdot \frac{1}{2N_l} \sum_{n=0}^{N_l-1} w[nN_c + \hat{\tau}_{lq} + mT_l] \cdot c_{lm}^{*}[n]\right\} \\
\tau_{lkqp}[m,m'] &\equiv \hat{\tau}_{lq} - \tau_{kp} + mT_l - m'T_k
\end{aligned} \tag{5}$$

In order to subtract interference we must, at a minimum, calculate $C_{lkqp}[m,m']$ for all virtual users and for all multipath components. A lower bound on the computational complexity can be determined by considering the above calculations for synchronous users. For synchronous users, all at the highest spreading factor, the required number of operations to calculate $C_{lkqp}[m,m']$ is $8(256)(2KL)^2 = 1.31$ Gops for $K = 100$ and $L = 4$. For real time operation 15000 such computations must be performed every second. This amounts to 19.7 TOPS (i.e. Tera Operations Per Second).

4. Regenerative MUD

Because of the extreme computational complexity of symbol-rate MUD for long codes it is advantageous to resort to regenerative MUD when long codes are used. Although regenerative MUD operates at the chip rate, the overall complexity is lower for long codes. For regenerative MUD the signal waveforms of interferers are regenerated at the sample rate and effectively subtracted from the received signal. A second pass through the matched filter then yields improved performance. It turns out that the computational complexity of regenerative MUD is linear in the number of users.

The received signal can be written

$$\begin{aligned} r[t] &= \sum_{k=1}^{2K}\sum_{m}\sum_{p=1}^{L} a_{kp} s_{km}[t-\tau_{kp} - mT_k]b_k[m] + w[t] \\ &= \sum_{k=1}^{2K} r_k[t] + w[t] \\ r_k[t] &\equiv \sum_{m}\sum_{p=1}^{L} a_{kp} s_{km}[t-\tau_{kp} - mT_k]b_k[m] \end{aligned} \tag{6}$$

Subtracting interference gives a cleaned-up signal $x_l[t]$

$$\begin{aligned} x_l[t] &= r[t] - \sum_{k=1,k\neq l}^{2K} \hat{r}_k[t] \\ &= r[t] - \sum_{k=1}^{2K} \hat{r}_k[t] + \hat{r}_l[t] \\ &= r[t] - \hat{r}[t] + \hat{r}_l[t] \\ &= \hat{r}_l[t] + r_{res}[t] \\ r_{res}[t] &\equiv r[t] - \hat{r}[t] \\ \hat{r}[t] &\equiv \sum_{k=1}^{2K} \hat{r}_k[t] \\ \hat{r}_k[t] &\equiv \sum_{m}\sum_{p=1}^{L} \hat{a}_{kp} s_{km}[t-\hat{\tau}_{kp} - mT_k]\hat{b}_k[m] \end{aligned} \tag{7}$$

Two methods are presented below for performing regenerative MUD.

First Method

In order to subtract interference we must reconstruct (regenerate) the waveform $s_{km}[t]$ as given in Equation (3). The waveform can be reconstructed using

$$
\begin{aligned}
s_{km}[t] &= \sum_{r=0}^{N_k-1} g[t - rN_c]c_{km}[r] \\
&= \sum_{p=0}^{N_k/4-1} \sum_{j=0}^{3} g[t-(4p+j)N_c]c_{km}[4p+j] \\
&= \sum_{p=0}^{N_k/4-1} \sum_{j=0}^{3} g[t-4pN_c - jN_c]c_{kmp}[j] \\
&= \sum_{p=0}^{N_k/4-1} s_{kmp}[t-4pN_c]
\end{aligned}
\tag{8}
$$

$$
s_{kmp}[t] \equiv \sum_{j=0}^{3} g[t-4pN_c - jN_c]c_{kmp}[j]
$$

$$
c_{kmp}[j] \equiv c_{km}[4p+j]
$$

The idea is that $s_{km}[t]$ can be represented as a summation of shifted waveforms $s_{kmp}[t]$, which are entirely specified by the 8 binary numbers comprising the complex sequence $c_{kmp}[j]$ of length 4. Hence there are only 2^8 = 256 such waveforms. For what follows we assume that the signals are sampled at N_c = 8 samples per chip. Each is of length 96 + 3(4) = 108 samples assuming that $g[t]$ is of length 96. For 2 bytes per sample (real and imaginary parts) the total memory requirement is 216*256 = 55296 bytes, which spills out of L1 cache, but fits entirely in L2 cache.

To generate $\hat{r}_k[t]$ for a single symbol period, 64 of these waveforms must be read from memory. For each of these 64 waveforms L complex macs are required per sample per symbol period. Hence 64(8L)(108) operations are required per symbol period. For L = 4 this amounts to 64(32)(108) = 221184 operations per symbol period (1/15 ms), or 3.32 GOPS. The formation of $r_{res}[t]$ then requires 2K times this, or 3.32(200) = 664 GOPS for K = 100 physical users. To form $\hat{r}_l[t]+r_{res}[t]$ requires an additional 2(96+255*4) = 2232 operations per symbol period per virtual user, or another 6.7 GOPS. Finally, the matched filter operation needs to be performed for each user, which from Equation (4) requires NLK complex macs (N = 256), or 256(4)(100)(8)*15000 = 12.3 GOPS. The GOPS figures above are for a single antenna. For two antennas the operations are doubled. Hence the total computational complexity is 2(664 + 6.7 + 12.3) = 1.37 TOPS. This is for a single-stage MPIC algorithm. For two stages the computation is doubled.

To perform regenerative MUD the baseband antenna stream data must be brought onto the MUD board. The required bandwidth is

[2 Bytes(complex)/Sa/Ant][2 Ant][8 Sa/chip][3.84 Mchips/ second] = 123 MB/s

Second Method

The second method is to represent the waveform for each multipath for each user as a complex impulse train with N_c = 8 samples per impulse. The complex amplitude of each impulse is the product of the complex chip, complex multipath amplitude and the binary (real) data bit estimate. These 2KL complex streams (times 2 for 2 antennas) are added to form a composite signal. Since this composite signal is a sum many impulse trains, all

asynchronous, the composite signal is a dense (i.e. no systematic zeros) signal at the sample rate. A block diagram of the processing is shown in Figure 1.

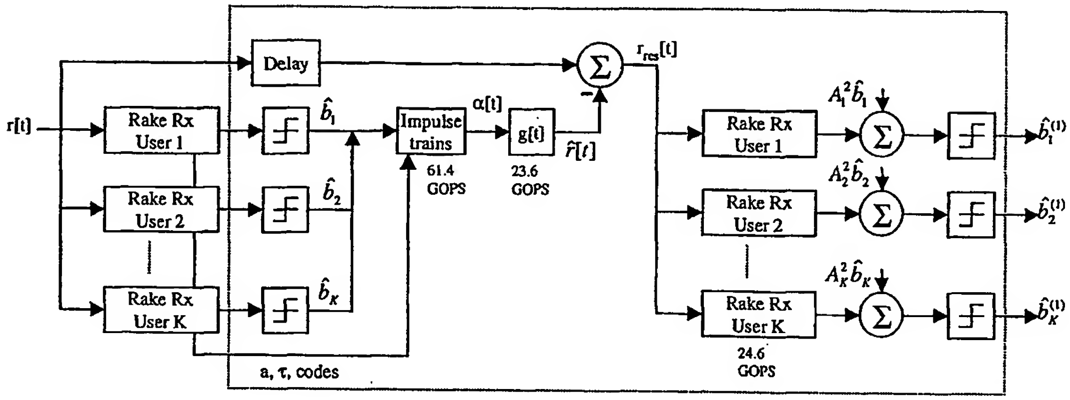


Figure 1. A block diagram of the long-code MUD processing

From Equations (7) and (8)

$$
\begin{aligned}
r_{res}[t] &\equiv r[t]-\hat{r}[t]\\
\hat{r}[t] &\equiv \sum_{k=1}^{2K}\hat{r}_k[t]\\
&=\sum_{k=1}^{2K}\sum_{m}\sum_{p=1}^{L}\hat{a}_{kp}s_{km}[t-\hat{\tau}_{kp}-mT_k]\hat{b}_k[m]\\
&=\sum_{k=1}^{2K}\sum_{p=1}^{L}\hat{a}_{kp}\sum_{m}\sum_{r=0}^{N_k-1}g[t-\hat{\tau}_{kp}-mT_k-rN_c]c_{km}[r]\hat{b}_k[m]\\
&=\sum_{k=1}^{2K}\sum_{p=1}^{L}\hat{a}_{kp}\sum_{m}\sum_{r=0}^{N_k-1}g[t-\hat{\tau}_{kp}-(r+mN_k)N_c]c_k[r+mN_k]\hat{b}_k[\lfloor (r+mN_k)/N_k\rfloor]\\
&=\sum_{k=1}^{2K}\sum_{p=1}^{L}\hat{a}_{kp}\sum_{n}g[t-\hat{\tau}_{kp}-nN_c]c_k[n]\hat{b}_k[\lfloor n/N_k\rfloor]\\
&=\sum_{k=1}^{2K}\sum_{p=1}^{L}\hat{a}_{kp}\sum_{r}\sum_{n}g[r]\delta[t-r-\hat{\tau}_{kp}-nN_c]c_k[n]\hat{b}_k[\lfloor n/N_k\rfloor]\\
&=\sum_{k=1}^{2K}\sum_{r}g[r]\sum_{p=1}^{L}\sum_{n}\delta[t-r-\hat{\tau}_{kp}-nN_c]\cdot\hat{a}_{kp}\cdot c_k[n]\cdot\hat{b}_k[\lfloor n/N_k\rfloor]\\
&=\sum_{r}g[r]\alpha[t-r]
\end{aligned}
\tag{9}
$$

$$
\alpha[t]\equiv\sum_{k=1}^{2K}\sum_{p=1}^{L}\sum_{n}\delta[t-\hat{\tau}_{kp}-nN_c]\cdot\hat{a}_{kp}\cdot c_k[n]\cdot\hat{b}_k[\lfloor n/N_k\rfloor]
$$

where *α[t]* is the composite signal. For each symbol period this requires 256(10)(2*KL*) operations per antenna. For two antennas this amounts to 5120(200)(4) = 4096000 operations per symbol period, or 61.4 GOPS.

The estimate of the received signal is then determined by passing the composite signal through the raised-cosine filter *g[t]* of length 96, which requires 96 real macs, or 192 real operations, per sample per real stream. There are a total of 4 real streams (2 antennas, real and imaginary streams). The total GOPS then for N_c = 8 samples per chip is 192(4)(8)(3.84M) = 23.6 GOPS.

The final step is to pass the cleaned-up signal $x_l[t] = \hat{r}_l[t] + r_{res}[t]$ through the matched-filter (i.e. rake receiver) which gives the improved detection statistic

$$
\begin{aligned}
y_l^{(1)}[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1} x_l[nN_c + \hat{\tau}_{lq} + mT_l]\cdot c_{lm}^*[n]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1} \hat{r}_l[nN_c + \hat{\tau}_{lq} + mT_l]\cdot c_{lm}^*[n]\right\} \\
&\quad + \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1} r_{res}[nN_c + \hat{\tau}_{lq} + mT_l]\cdot c_{lm}^*[n]\right\} \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1}\left[\sum_{m'}\sum_{q'=1}^{L} \hat{a}_{lq'} s_{lm'}[nN_c + \hat{\tau}_{lq} - \hat{\tau}_{lq'} + (m-m')T_l]\hat{b}_l[m']\right]\cdot c_{lm}^*[n]\right\} + y_{l,res}^{(1)}[m] \\
&\cong \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1}\left[\sum_{q'=1}^{L} \hat{a}_{lq'} s_{lm}[nN_c + \hat{\tau}_{lq} - \hat{\tau}_{lq'}]\right]\cdot c_{lm}^*[n]\right\}\cdot \hat{b}_l[m] + y_{l,res}^{(1)}[m] \\
&= \mathrm{Re}\left\{\sum_{q=1}^{L}\sum_{q'=1}^{L} \hat{a}_{lq}^H \hat{a}_{lq'} \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1} s_{lm}[nN_c + \hat{\tau}_{lq} - \hat{\tau}_{lq'}]\cdot c_{lm}^*[n]\right\}\cdot \hat{b}_l[m] + y_{l,res}^{(1)}[m] \\
&\cong \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \hat{a}_{lq}\right\}\cdot \hat{b}_l[m] + y_{l,res}^{(1)}[m] \\
&= A_l^2 \cdot \hat{b}_l[m] + y_{l,res}^{(1)}[m]
\end{aligned}
$$

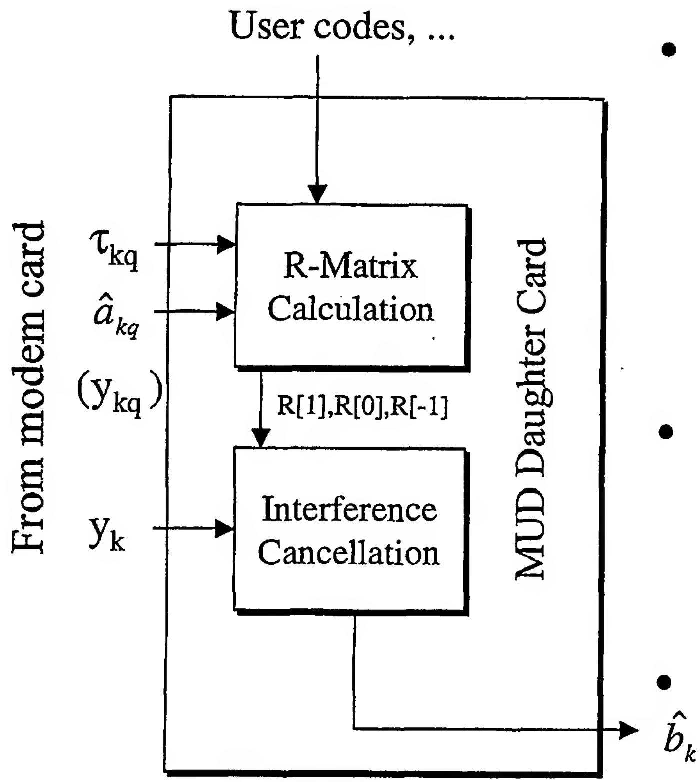
$$
\begin{aligned}
A_l^2 &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \hat{a}_{lq}\right\} \\
y_{l,res}^{(1)}[m] &\equiv \mathrm{Re}\left\{\sum_{q=1}^{L} \hat{a}_{lq}^H \cdot \frac{1}{2N_l}\sum_{n=0}^{N_l-1} r_{res}[nN_c + \hat{\tau}_{lq} + mT_l]\cdot c_{lm}^*[n]\right\}
\end{aligned}
\tag{10}
$$

The matched filter operation requires *NLK* complex macs, or 256(4)(100)(8)*15000 = 12.3 GOPS. The GOPS figures above are for a single antenna. For two antennas the operations are doubled, giving 24.6 GOPS. The total computational complexity for the second method is then 61.4 + 23.6 + 24.6 = 109.6 GOPS.

References

[1] J. H. Oates, "MUD Algorithms," Mercury Wireless Communication Group Report, August 22, 2000.

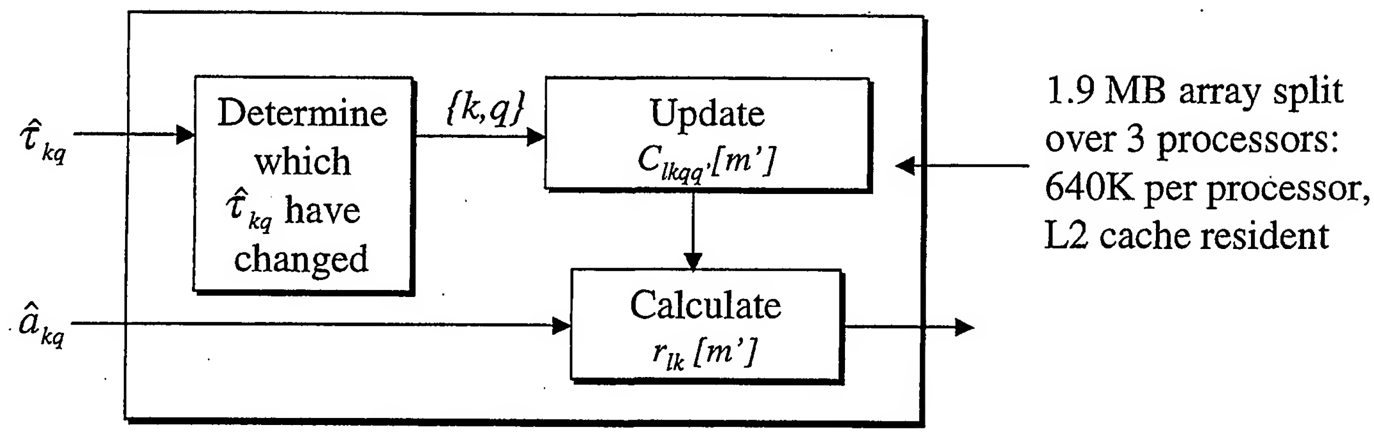
MUD Functions



- R-matrix calculation
  - User code correlations
    - C-matrices (1.9 MB)
    - τ_{kq} change ~ every 100 ms
  - R-matrices
    - a_{kq} change ~ every 1.33 ms
- Interference cancellation
  - Must be performed in 10 symbol periods (0.667 ms) for real-time operation
- Total latency: 10 ms

To symbol processing

R-matrix Calculation



$$r_{lk}[m'] \equiv \rho_{lk}[m'] A_l A_k \equiv \sum_{q=1}^{L} \sum_{q'=1}^{L} \mathrm{Re}\left\{ \hat{a}_{lq}^{H} \hat{a}_{kq'} \cdot C_{lkqq'}[m'] \right\}$$

$$C_{lkqq'}[m'] \equiv \frac{1}{2N_l} \sum_{n} s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}] \cdot c_l^*[n]$$

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: R-matrix GOPS

Date: June 21, 2000

1. Introduction

This report investigates a number of different methods for calculating the R-matrix elements. There are two parts to the calculation. First is the calculation of the user code correlations at lag offsets determined by the searcher receivers. This calculation must be performed every time a multipath component changes to a new lag. The assumption used here is that every 100 ms one multipath component changes to a new lag for each user. Hence, if each user has 4 multipath lags, then all R-matrix elements will have changed after 400 ms. The validity of this assumption will have to be tested with measured data. Note that the WCDMA standard call out a test with 2 multipath components, where one lag changes every 191 ms [1]. The second part is the actual calculation of the R-matrix elements, which requires a double summation of code correlations over all multipath components, with each term scaled by the Rayleigh-fading multipath amplitudes. The maximum time period to perform this calculation is about 1.33 ms. Hence there are two parts to the calculation, each with a different update rate.

Section 2 is devoted the first part of the calculation, the code correlations. Section 3 covers the actual calculation of the R-matrix elements.

2. Calculation of User Code Correlations

The R-matrix elements can be expressed as [2]

$$\hat{\rho}_{lk}[m']A_k = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{*}\hat{a}_{kq'}\cdot C_{lkqq'}[m']\right\}$$

$$C_{lkqq'}[m'] \equiv \frac{1}{2N_l}\sum_{n} s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^{*}[n] \tag{1}$$

$$= \frac{1}{2N_l}\sum_{n}\sum_{p} g[(n-p)N_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]c_k[p]\cdot c_l^{*}[n]$$

where $C_{lkqq'}[m']$ is a five-dimensional matrix of code correlations. Both l and k range from 1 to K_v, where K_v is the number of virtual users. If there are K physical users, all operating at the highest spreading factor, then there are $K_v = 2K$ virtual users. For now consider $K = 128$ so that $K_v = 256$. The indices q and q' range from 1 to L, the number of multipath components, which for this report is assumed to be equal to 4. The symbol period offset m' ranges from –1 to 1. The total number of matrix elements to be calculated is then $N_C = 3(K_v L)^2 = 3(1024)^2 = 3M$ complex elements, or 24 MB if each element is a float. This number is reduced, however, due to the symmetries

$$C_{klq'q}[-m'] = \frac{1}{2N_k}\sum_{n}\sum_{p} g[(n-p)N_c - m'T + \hat{\tau}_{kq'} - \hat{\tau}_{lq}]c_l[p]\cdot c_k^{*}[n]$$

$$= \frac{1}{2N_k}\sum_{p}\sum_{n} g[-(n-p)N_c - m'T - \hat{\tau}_{lq} + \hat{\tau}_{kq'}]c_l[n]\cdot c_k^{*}[p] \tag{2}$$

$$= \frac{1}{2N_k}\sum_{p}\sum_{n} g[(n-p)N_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]c_k^{*}[p]\cdot c_l[n]$$

$$= \frac{N_l}{N_k}C_{lkqq'}^{*}[m']$$

so that it is sufficient to store elements for offsets $m' = 0,1$. The memory requirement is then 16 MB if each element is a float. If the elements are stored as bytes the requirement is reduced to 4 MB.

Referring to Equation 1, line 2, it is evident that each element of $C_{lkqq'}[m']$ is a complex dot product between a code vector c_l and a waveform vector $s_{kqq'}$. The length of the code vector is 256. The length of the waveform vector is $L_g + 255N_c$, where L_g is the length of the raised-cosine pulse vector $g[t]$ and N_c is the number of samples per chip. The values for these parameters as currently implemented are $L_g = 48$ and $N_c = 4$. The length of the waveform vector is then 1068, but for the dot product it is accessed at a stride of $N_c = 4$, which gives effectively a length of 267. Note that the code and waveform vectors in general do not entirely overlap. Also note that an increment or decrement in the symbol offset index m' slides the waveform vector 256 elements to the left or right respectively. Figure 1 shows that the total number of complex macs (cmacs) for all three ($m' = -1, 0, 1$) dot products is 267, irrespective of any relative offset.

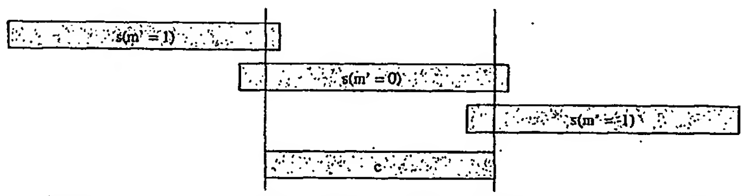


Figure 1. Overlap of waveform and code vectors. The total number of complex macs (cmacs) for all three (m' = -1, 0, 1) dot products is 267, irrespective of any relative offset.

Hence for any given combination of indices $lkqq'$ the three elements $C_{lkqq'}[m']$, corresponding to $m' = -1$, 0 and 1 require 267 cmacs to calculate all three. Since there are $(K_vL)^2$ combinations of indices, the calculation of all elements $C_{lkqq'}[m']$ requires $(K_vL)^2$ (267) cmacs. Given the symmetry condition, only half of the elements need to be calculated, and noting that each cmac requires 8 operation to perform; the total number of operations required is

$$N_{ops} = \tfrac{1}{2}(K_vL)^2(267)(8) = \tfrac{1}{2}(1024)^2(267)(8) = 1.12\ G\ ops \qquad (3)$$

The total number of GOPS (Giga Operations Per Second), then, given the 400 ms update rate is

$$N_{GOPS} = \frac{\tfrac{1}{2}(K_vL)^2(267)(8)ops}{400ms} = \frac{\tfrac{1}{2}(1024)^2(267)(8)ops}{400ms} = 2.80\ GOPS \qquad (4)$$

The next section addresses the calculation of the R-matrix elements.

3. Calculation of R-matrix Elements

Consider the calculation of the R-matrix elements

$$\hat{\rho}_{lk}[m']A_k = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\{\hat{a}_{lq}^{*}\hat{a}_{kq'}\cdot C_{lkqq'}[m']\} \quad (5)$$

The total number of matrix elements to be calculated is $N_\rho = 3K_v^2$. This number is reduced, however, due to the symmetries

$$\begin{aligned}\hat{\rho}_{kl}[-m']A_l &= \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\{\hat{a}_{kq}^{*}\hat{a}_{lq'}\cdot C_{klqq'}[-m']\} = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}\hat{a}_{kq'}^{*}\cdot\frac{N_l}{N_k}C_{lkqq'}^{*}[m']\right\} \\ &= \frac{N_l}{N_k}\sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\left[\hat{a}_{lq}^{*}\hat{a}_{kq'}\cdot C_{lkqq'}[m']\right]^{*}\right\} = \frac{N_l}{N_k}\hat{\rho}_{lk}[m']A_k\end{aligned} \quad (6)$$

so that the total number of matrix elements to be calculated is $N_\rho = \frac{3}{2}K_v^2$.

Now let us consider the operations per element. Dropping explicit reference to the symbol period offset $[m']$, the matrix elements are

$$\hat{\rho}_{lk}A_k = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\{\hat{a}_{lq}^{*}\cdot\hat{a}_{kq'}\cdot C_{lkqq'}\} \quad (7)$$

A brute-force calculation requires $L^2(6 + 3 + 1)$ operations (1 complex multiply, one half-complex multiply -- i.e. the real part -- and one real add, or 6 real multiplies and 4 real adds). The total operations is then

$$N_{ops} = \tfrac{3}{2}(K_v L)^2(10) \quad (8)$$

For a vehicular speed of 120 km/h the Doppler frequency is 216.67 Hz for a user at frequency 1950 MHz. The coherence bandwidth is thus 433.33 MHz, and the corresponding coherence time is about 2.3 ms. Hence the multipath amplitudes are changing with a time constant of about 2 ms, and consequently the second part of the calculation must be updated at least every 2 ms. The channel amplitudes are calculated on a time slot by time slot basis. Each time slot is 10/15 = 2/3 = 0.67 ms. Hence 2 ms equals 3 time slots, whereas two slots equals 1.33 ms. Figures 2 and 3 below show the MUD efficiency versus user velocity for 2 ms and 1.33 ms update times respectively. The plots show that to be able to effectively handle high velocity users the update time should be 1.33 ms. When users are at various speeds the interference from low speed users is cancelled more effectively than the interference from high speed users. The MUD efficiency

will then be an average of the MUD efficiency corresponding to each user's speed.

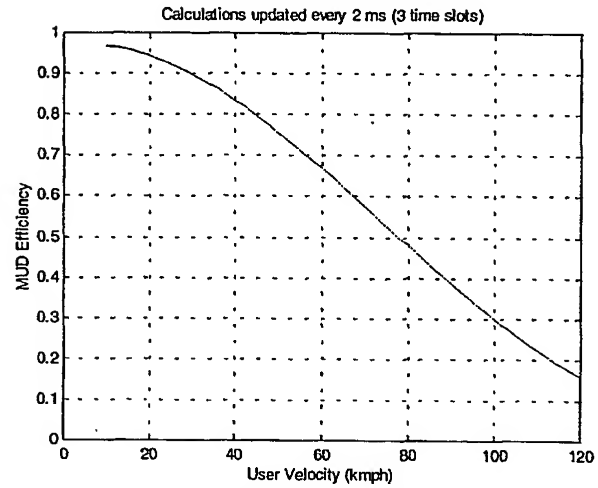


Figure 2. MUD efficiency versus user velocity for a 2 ms R-matrix update time.

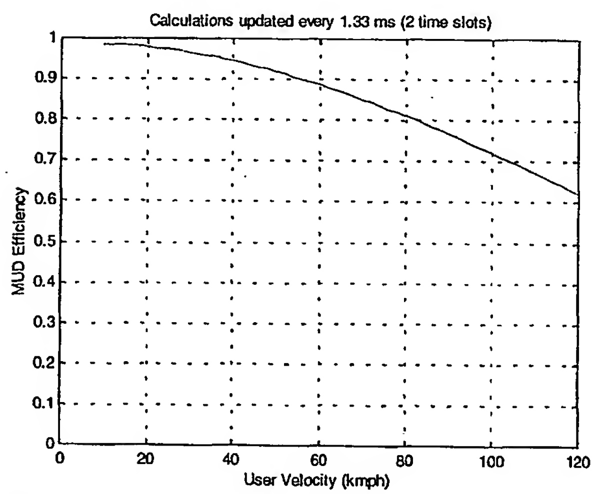


Figure 3. MUD efficiency versus user velocity for a 1.33 ms R-matrix update time.

The calculations below are based on a 1.33 ms update time. Note that most of the capacity and coverage benefits calculated for MUD so far have assumed

70% MUD efficiency. The 1.33 ms update time is sufficient to achieve 70% MUD efficiency. The total GOPS are then,

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2(10)}{1.33\,ms} = \frac{1.5(256 \cdot 4)^2(10)}{1.33\,ms} = 11.8\,GOPS \tag{9}$$

where we have assumed $L = 4$ multipath components. A better way to perform this operation is

$$\hat{\rho}_{lk} A_k = \sum_{q=1}^{L} \mathrm{Re}\left\{ \hat{a}_{lq}^{*} \sum_{q'=1}^{L} C_{lkqq'} \cdot \hat{a}_{kq'} \right\} \tag{10}$$

The inner sum is a matrix-vector multiply, hence requiring L^2 cmacs, and the outer sum is the real part of a compex dot product, which requires L half-cmacs. The total is then $(L^2 + L/2) = 1.125\,L^2$ cmacs (for $L = 4$) times 8 operations per cmac, or $9L^2$ operations, which gives

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2(9)}{1.33\,ms} = \frac{1.5(256 \cdot 4)^2(9)}{1.33\,ms} = 10.6\,GOPS \tag{11}$$

The above calculations are represented in terms of complex numbers, which are not directly calculable. To express the above equations explicitly in terms of real numbers it is convenient to cast the calculations into matrix form

$$\begin{aligned}
\hat{\rho}_{lk} A_k &= \sum_{q=1}^{L} \sum_{q'=1}^{L} \mathrm{Re}\left\{ \hat{a}_{lq}^{*} \hat{a}_{kq'} \cdot C_{lkqq'} \right\} \\
&= \mathrm{Re}\left\{ \begin{bmatrix} \hat{a}_{l1}^{*} & \hat{a}_{l2}^{*} & \cdots & \hat{a}_{lL}^{*} \end{bmatrix} \begin{bmatrix} C_{lk11} & C_{lk12} & \cdots & C_{lk1L} \\ C_{lk21} & C_{lk22} & \cdots & C_{lk2L} \\ \vdots & \vdots & \ddots & \vdots \\ C_{lkL1} & C_{lkL2} & \cdots & C_{lkLL} \end{bmatrix} \begin{bmatrix} \hat{a}_{k1} \\ \hat{a}_{k2} \\ \vdots \\ \hat{a}_{kL} \end{bmatrix} \right\} \\
&\equiv \mathrm{Re}\left\{ a_l^H \cdot C_{lk} \cdot a_k \right\}
\end{aligned} \tag{12}$$

$$a_k \equiv \begin{bmatrix} \hat{a}_{k1} \\ \hat{a}_{k2} \\ \vdots \\ \hat{a}_{kL} \end{bmatrix}, \qquad C_{lk} \equiv \begin{bmatrix} C_{lk11} & C_{lk12} & \cdots & C_{lk1L} \\ C_{lk21} & C_{lk22} & \cdots & C_{lk2L} \\ \vdots & \vdots & \ddots & \vdots \\ C_{lkL1} & C_{lkL2} & \cdots & C_{lkLL} \end{bmatrix}$$

The quadratic form $a_l^H C_{lk} a_k$ can be expressed

$$\mathrm{Re}\{a_l^H \cdot C_{lk} \cdot a_k\} = \mathrm{Re}\{[a_r^T - ja_i^T] \cdot [C_r + jC_i] \cdot [b_r + jb_i]\}$$

$$= \mathrm{Re}\{[a_r^T - ja_i^T] \cdot [C_r b_r - C_i b_i + j(C_r b_i + C_i b_r)]\}$$

$$= \mathrm{Re}\begin{Bmatrix} a_r^T C_r b_r - a_r^T C_i b_i + a_i^T C_r b_i + a_i^T C_i b_r \\ + j(a_r^T C_r b_i + a_r^T C_i b_r - a_i^T C_r b_r + a_i^T C_i b_i) \end{Bmatrix}$$

$$= \mathrm{Re}\left\{ \begin{bmatrix} a_r^T & a_i^T \end{bmatrix} \begin{bmatrix} C_r & -C_i \\ C_i & C_r \end{bmatrix} \begin{bmatrix} b_r \\ b_i \end{bmatrix} + j \begin{bmatrix} a_r^T & a_i^T \end{bmatrix} \begin{bmatrix} C_i & C_r \\ -C_r & C_i \end{bmatrix} \begin{bmatrix} b_r \\ b_i \end{bmatrix} \right\}$$

$$= \begin{bmatrix} a_r^T & a_i^T \end{bmatrix} \begin{bmatrix} C_r & -C_i \\ C_i & C_r \end{bmatrix} \begin{bmatrix} b_r \\ b_i \end{bmatrix}$$

$$\equiv a^T \cdot C \cdot b \tag{13}$$

The matrix-vector multiplication requires $(2L)^2$ macs. The dot product adds $(2L)$ macs so that the total is $(2L)^2 + (2L)$ macs. For $L = 4$ we have $1.125(2L)^2$ macs = $4.5L^2$ macs = $9L^2$ operations. The total GOPS are then

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2 (9)}{1.33\,ms} = \frac{1.5(256 \cdot 4)^2 (9)}{1.33\,ms} = 10.6\,GOPS \tag{14}$$

Now consider a different formulation which attempts to reuse the amplitude-amplitude multiplications. Consider the calculation $a^T \cdot C \cdot b$

$$a^T \cdot C \cdot b = tr[a^T \cdot C \cdot b] = tr[C \cdot (ba^T)] = tr[C \cdot X]$$
$$X \equiv ba^T \tag{15}$$

The calculations to produce matrix X are pure multiplications, but the elements, once calculated, can be reused for the other virtual users corresponding to the same physical users. For voice-only users there are 2 virtual users per physical user. For data users there can be up to 65 virtual users per physical user. For now, however, we stay with our 128 voice-user scenario. To calculate X, then, requires $(2L)^2 = 4L^2$ multiplications. This calculation is performed once per pair of physical users, so the total number of operations is

$$N_{ops} = (KL)^2 (4) = (K_v L)^2 (1) = \tfrac{3}{2}(K_v L)^2 (\tfrac{2}{3}) \tag{16}$$

Effectively, then, X requires $(2/3)L^2$ operations. The details to calculate $a^T \cdot C \cdot b$ are

$$a^T \cdot C \cdot b = tr[C \cdot X] = tr\left\{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_L \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & \cdots & x_L \end{bmatrix}\right\} = \sum_{i=1}^{2L} c_i \cdot x_i \tag{17}$$

where c_i is the ith row of C and x_i is the ith column of X. Hence we have $2L$ dot products of length $2L$, which require $(2L)^2$ macs = $8L^2$ operations. To calculate $a^H \cdot C \cdot b$ then requires $8L^2 + (2/3)L^2 = 8.67L^2$ operations, which gives

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2 (8.67)}{1.33\ ms} = \frac{1.5(256 \cdot 4)^2 (8.67)}{1.33\ ms} = 10.3\ GOPS \tag{18}$$

A better way to perform this calculation is as follows

$$\rho = \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\{a_q^* \cdot a_{q'} \cdot C_{qq'}\} = \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\{X_{qq'}^* \cdot C_{qq'}\}$$

$$= \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\{(X_{qq'}^r - jX_{qq'}^i) \cdot (C_{qq'}^r + jC_{qq'}^i)\}$$

$$= \sum_{q=1}^{L}\sum_{q'=1}^{L} (X_{qq'}^r \cdot C_{qq'}^r + X_{qq'}^i \cdot C_{qq'}^i) \tag{19}$$

$$X_{qq'} \equiv a_q \cdot a_{q'}^* = (a_q^r + ja_q^i) \cdot (a_{q'}^r - ja_{q'}^i) \equiv X_{qq'}^r + jX_{qq'}^i$$
$$X_{qq'}^r = a_q^r \cdot a_{q'}^r + a_q^i \cdot a_{q'}^i$$
$$X_{qq'}^i = a_q^i \cdot a_{q'}^r - a_q^r \cdot a_{q'}^i$$

where for convenience we have dropped A_k, the lk subscripts and the hat symbols. The calculation of X requires

$$N_{ops} = (KL)^2(6) = (K_v L)^2 (6/4) = \tfrac{3}{2}(K_v L)^2 (1) \tag{20}$$

operations. Note that, once the X values are calculated, the remainder of the calculation is a long dot product of length $2L^2$, hence requiring $2L^2$ macs, or $4L^2$ operations, which gives

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2(5)}{1.33\ ms} = \frac{1.5(256\cdot 4)^2(5)}{1.33\ ms} = 5.9\ GOPS \tag{21}$$

Dual Diversity Antennas

When dual diversity antennas are employed, the calculation of the R-matrix elements becomes

$$\begin{aligned}
\rho &= \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\left\{\left(a_{1q}^{*}\cdot a_{1q'} + a_{2q}^{*}\cdot a_{2q'}\right)\cdot C_{qq'}\right\} = \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\left\{\left(X_{1qq'}^{*} + X_{2qq'}^{*}\right)\cdot C_{qq'}\right\} \\
&= \sum_{q=1}^{L}\sum_{q'=1}^{L}\left[\left(X_{1qq'}^{r} + X_{2qq'}^{r}\right)\cdot C_{qq'}^{r} + \left(X_{1qq'}^{i} + X_{2qq'}^{i}\right)\cdot C_{qq'}^{i}\right] \\
&= \sum_{q=1}^{L}\sum_{q'=1}^{L}\left[X_{qq'}^{r}\cdot C_{qq'}^{r} + X_{qq'}^{i}\cdot C_{qq'}^{i}\right]
\end{aligned} \tag{22}$$

$$X_{qq'}^{r} \equiv X_{1qq'}^{r} + X_{2qq'}^{r}$$
$$X_{qq'}^{i} \equiv X_{1qq'}^{i} + X_{2qq'}^{i}$$

To calculate X for dual diversity antennas, then, requires

$$N_{ops} = (KL)^2(14) = (K_v L)^2(14/4) = \tfrac{3}{2}(K_v L)^2(7/3) = \tfrac{3}{2}(K_v L)^2(2.33) \tag{23}$$

operations. The remainder of the calculation is again a long dot product of length $2L^2$ requiring $4L^2$ operations, which gives

$$N_{GOPS} = \frac{\frac{3}{2}(K_v L)^2(6.33)}{1.33\ ms} = \frac{1.5(256\cdot 4)^2(6.33)}{1.33\ ms} = 7.5\ GOPS \tag{24}$$

Reuse of C data

So far we have not addressed the problem associated with a lack of data reuse, which renders our calculations I/O limited. The C data can be reused by introducing extra latency into the calculations. For a given user, a single multipath component changes on average once every 100 ms, or once every 150 slots. Suppose we collect and save in cache 4 amplitude estimate vectors $a_k[q]$,

where q is the 2 ms update index. The total latency is then 8 ms = 12 time slots. During this time the probability that a multipath lag changes is (8 ms)/(100 ms) = .08. The probability that the matrix C_{lk} changes is then = $1-(1-0.08)^2 = 0.15$. Hence for most matrices C_{lk} we will be able to calculate

$$a_l^H[q] \cdot C_{lk} \cdot a_k[q] \tag{25}$$

for 12 time slots q for only one read of C_{lk} from memory. The penalty for this reuse is the 8 ms of latency incurred.

References

[1] "3rd Generation Partnership Project (3GPP) Technical Specification Group (TSG) RAN WG4 UE Radio transmission and Reception (FDD)", TS 25.101 V3.1.0 (1999-12), Annex B.

[2] J. H. Oates, "MUD Algorithms," Mercury Wireless Communications Group Report, April 25, 2000.

- MERCURY COMPUTER SYSTEMS PROPRIETARY INFORMATION –

MERCURY Computer Systems, Inc.
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Memorandum

To: John Oates, John Greene, Alden Fuchs, Frank Lauginiger — Date: 31-AUG-2000

From: Mike Vinskus

Subject: Theoretically optimum load balancing for the R matrix calculations — File Ref: mjv-9.doc

This memo describes the calculation of optimum R matrix partitioning points in normalized virtual user space. These partitioning points provide an equal, and hence balanced, computation load per processor. The computational model of the R matrix calculations does not include any data access overhead or caching effects. It is shown that a closed form recursive solution exists that can be solved for an arbitrary number of processors.

Although three R matrices are output from the R matrix calculation function, only half of the elements are explicitly calculated. This is due to the symmetry condition that exists between R matrices:

$$R_{l,k}(m) = \xi R_{k,l}(-m).$$

In essence, only two matrices need to be calculated. The first one is a combination of R(1) and R(-1). The second is the R(0) matrix. In this case, the essential R(0) matrix elements have a triangular structure to them. The number of computations performed to generate the raw data for the R(1)/R(-1) and R(0) matrices are combined and optimized as a single number. This is due to the reuse of the X matrix outer product values across the two R matrices. Since the bulk of the computations involve combining the X matrix and correlation values, they dominate the processor utilization. These computations are used as a cost metric in determining the optimum loading of each processor.

The optimization problem is formulated as an equal area problem, where the solution results in each partition area to be equal. Since the major dimensions of the R matrices are in terms of the number of active virtual users, the solution space for this problem is in terms of the number of virtual users per processor. By normalizing the solution space by the number of virtual users, the solution is applicable for an arbitrary number of virtual users.

- MERCURY COMPUTER SYSTEMS PROPRIETARY INFORMATION –

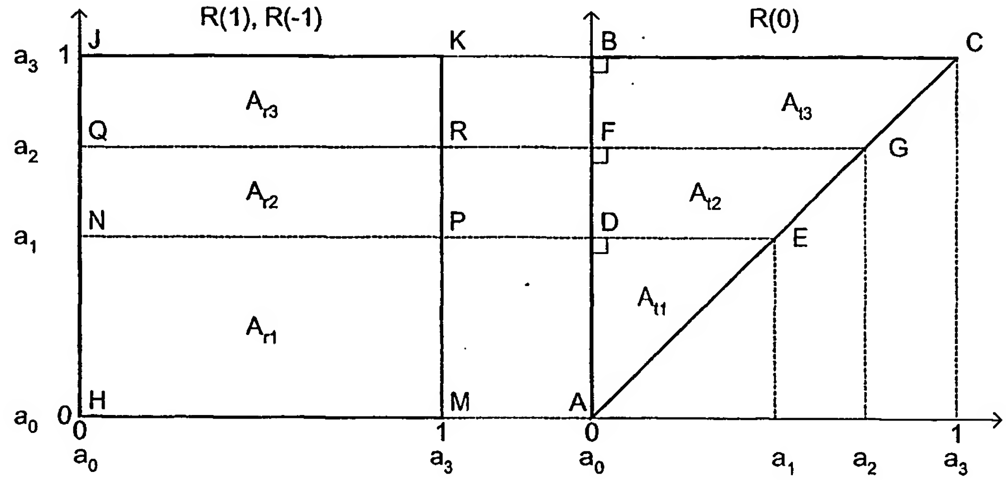


Figure 1: Normalized R matrix computation model.

Figure 1 shows the model of the normalized optimization problem. The computations for the R(1)/R(-1) matrix are represented by the square HJKM, while the computations for the R(0) matrix are represented by the triangle ABC. From geometry, the area of a rectangle of length b and height h is

$$A_r = bh\,.$$

For a triangle with a base width b and height h, the area is calculated by

$$A_t = \frac{1}{2}bh\,.$$

When combined with a common height a_i, the formula for the area becomes

$$\begin{aligned} A_i &= A_{ri} + A_{ti} \\ &= a_i a_3 + \frac{1}{2}a_i^2\,. \\ &= a_i + \frac{1}{2}a_i^2 \end{aligned}$$

The formula for A_i gives the area for the total region below the partition line. For example, the formula for A2 gives the area within the rectangle HQRM plus the region within triangle AFG. For the cost function, the difference in successive areas is used. That is

$$\begin{aligned} B_i &= A_i - A_{i-1} \\ &= \frac{1}{2}a_i^2 + a_i - \frac{1}{2}a_{i-1}^2 - a_{i-1} \end{aligned}$$

For an optimum solution, the B_i must be equal for $i = 1, 2, ..., N$, where N is the number of processors performing the calculations. Because the total normalized load is equal to A_N, the loading per processor load is equal to A_N/N.

$$B_i = \frac{A_N}{N} = \frac{A_3}{3} = \frac{3}{2N}, \text{ for i} = 1, 2, ..., N.$$

By combining the two equation for B_i, the solution for a_i is found by finding the roots of the equation:

$$\frac{1}{2}a_i^2 + a_i - \frac{1}{2}a_{i-1}^2 - a_{i-1} - \frac{3}{2N} = 0.$$

The solution for a_i is:

$$a_i = -1 \pm \sqrt{1 + a_{i-1}^2 + 2a_{i-1} + \frac{3}{N}}, \text{ for } i = 1, 2, ..., N.$$

Since the solution space must fall in the range [0, 1], negative roots are not valid solutions to the problem. On the surface, it appears that the a_i must be solved by first solving for case where $i = 1$. However, by expanding the recursions of the a_i and using the fact that a_0 equals zero, a solution that does not require previous a_i, $i = 0, 1, ..., n-1$ exists. The solution is:

$$a_i = -1 + \sqrt{1 + \frac{3i}{N}}$$

Table 1 shows the normalized partition values for two, three, and four processors. To calculate the actual partitioning values, the number of active virtual users is multiplied by the corresponding table entries. Since a fraction of a user cannot be allocated, a ceiling operation is performed that biases the number of virtual users per processor towards the processors whose loading function is less sensitive to perturbations in the number of users.

Table 1: Normalized partition locations for two, three, and four processors.

Location	Two processors	Three processors	Four processors
a_1	$-1+\sqrt{\frac{5}{2}}$ (0.5811)	$-1+\sqrt{2}$ (0.4142)	$-1+\sqrt{\frac{7}{4}}$ (0.3229)
a_2	--	$-1+\sqrt{3}$ (0.7321)	$-1+\sqrt{\frac{5}{2}}$ (0.5811)
a_3	--	--	$-1+\sqrt{\frac{13}{4}}$ (0.8028)

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Memorandum

To: Jonathan Schonfeld

Date: 23-FEB-2001

From: Nmf

Subject: Degraded mode of operation for the MUD algorithm

File Ref: mjv-018-degraded_mode_desc.doc

Reference [1] showed that the load balancing for the R matrix calculations resulted in a non -uniform partitioning of the rows of the final R matrices over a number of processors. In summary, the partition sizes increase as the partition starting user index increases.

When the system is running at full capacity (i.e. the maximum number of users is processed while still within the bounds of real-time operation) and a computational node has a failure, the impact can be significant.

This impact can be minimized by allocating the first user partition to the disabled node. Also the values that would have been calculated by that node are set to zero. This reduces the effects of the failed node. Also, by changing which user data is set to zero (i.e. which users are assigned to the failed node) the overall errors due to the lack of non-zero output data for that node are averaged over all of the users, providing a "soft" degradation.

References:

[1] M. Vinskus. "mjv-009: Theoretically optimum load balancing for the R matrix calculations." 31-AUG-2000.
[2] M. Vinskus. "mjv-010: Preliminary degraded MUD operation results." 19-OCT-2000.
[3] J. Oates. "jho-001: MUD Algorithms", 25-APR-2000

Mercury Computer Systems, Inc.
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: Methods for Calculating the C-matrix Elements Date: November 13, 2000

1. Direct Method

The direct method for calculating the C-matrix elements is

$$\hat{r}_{lk}[m'] = \sum_{q=1}^{L}\sum_{q'=1}^{L} \mathrm{Re}\left\{\hat{a}_{lq}^{*}\hat{a}_{kq'} \cdot C_{lkqq'}[m']\right\}$$

$$C_{lkqq'}[m'] \equiv \frac{1}{2N_l}\sum_{n} s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}] \cdot c_l^{*}[n] \tag{1}$$

Symmetry

$$C_{klq'q}[-m'] = \frac{N_l}{N_k} C_{lkqq'}^{*}[m'] \tag{2}$$

Due to symmetry there are $1.5(K_vL)^2$ elements to calculate. Assuming all users are at SF 256, each calculation requires 256 cmacs, or 2048 operations. The probability that a multipath changes in a 10 ms time period is approximately 10/200 = 0.05 if all users are at 120 kmph. Assuming a mix of user velocities, let's say the probability is 0.025. Since the C-matrix elements represent the interaction between two users, the probability that C-matrix elements change in a 10 ms time period is approximately 0.10 for all users are at 120 kmph, or 0.05 for a mix of user velocities. The GOPS are tabulated in Table 1 below.

The C-matrix elements also need to be updated when the spreading factor changes. The spreading factor can change due to

- AMR codec rate changes
- Multiplexing of DCCH
- Multiplexing data services

For lack of a better number, assume that 5% of the users, hence 10% of the elements change rate every 10 ms.

Table 1. GOPS to update C-matrix elements using the direct method.

K_v	High velocity users	$1.5(K_vL)^2$	Gops	Percentage change	GOPS
200	100%	960,000	1.966	20	39.3
200	50%	960,000	1.966	15	29.5
128	100%	393,216	0.805	20	16.1
128	50%	393,216	0.805	15	12.1

2. FFT Method

The FFT can be used to calculate the correlations for a range of offsets τ using

$$\begin{aligned} C_{lkqq'}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^*[n] \\ &= C_{lk}[\tau_{lkqq'}[m']] \end{aligned} \tag{3}$$

$$C_{lk}[\tau] \equiv \frac{1}{2N_l}\sum_n s_k[nN_c + \tau]\cdot c_l^*[n]$$

$$\tau_{lkqq'}[m'] \equiv m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}$$

The length of the waveform $s_k[t]$ is $L_g + 255N_c = 1068$ for $L_g = 48$ and $N_c = 4$. This is represented as N_c waveforms of length $L_g/N_c + 255 = 267$.

One advantage of this approach is that elements can be stored for a range of offsets τ so that calculations do not need to be performed when lags change. For delay spreads of about 4μs 32 samples need to be stored for each m'.

3. Using Code Correlations

The C-matrix elements can be represented in terms of the underlying code correlations using

$$\begin{aligned}
C_{lkqq'}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_p g[(n-p)N_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_k[p]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_m g[mN_c + \tau]\cdot c_k[n-m]\cdot c_l^*[n] \\
&= \sum_m g[mN_c + \tau]\cdot \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m] \qquad (4) \\
&= \sum_m g[mN_c + \tau]\cdot \Gamma_{lk}[m]
\end{aligned}$$

$$\Gamma_{lk}[m] \equiv \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m]$$

$$\tau \equiv m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}$$

If the length of *g[t]* is $L_g = 48$ and $N_c = 4$, then the summation over m requires 48/4 =12 macs for the real part and 12 macs for the imaginary part. The total ops is then 48 ops per element. (Compare with 2048 operations for the direct method.) Hence for the case where there are 200 virtual users and 20% of the C-matrix needs updating every 10 ms the required complexity is (960000 el)(48 ops/el)(0.20)/(0.010 sec) = 921.6 MOPS. This is the required complexity to compute the C-matrix from the Γ-matrix. The cost of computing the Γ-matrix must also be considered. There is reason to hope that the Γ-matrix can be efficiently computed since the fundamental operation is a convolution of codes with elements constrained to be +/-1 +/-j.

The Γ-matrix elements can be calculated using

- the FFT
- Modulo-2 arithmetic
- Hardware XOR
- Short-code generator(?)

4. Using Fundamental Correlations

The waveform $s_k[t]$ can be decomposed into fundamental waveforms corresponding to 4-chip segments of the corresponding complex user codes. There are $2^8 = 256$ such waveforms. Each of these can be correlated with another 256 possible 4-chip code segments. For each correlation there are about 64 offsets that produce a non-zero correlation. Hence all correlation calculations can be represented in terms of 256(256)(64) = 4M fundamental complex correlations. The C-matrix elements are then

$$\begin{aligned} C_{lkqq'}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^*[n] \\ &= C_{lk}[\tau_{lkqq'}[m']] \end{aligned}$$

$$C_{lk}[\tau] \equiv \frac{1}{2N_l}\sum_n s_k[nN_c + \tau]\cdot c_l^*[n]$$

$$\tau_{lkqq'}[m'] \equiv m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}$$

$$\begin{aligned} C_{lk}[\tau] &\equiv \sum_{i=0}^{63}\sum_{j=0}^{63}\frac{1}{2N_l}\sum_{n=0}^{3} s_{kj}[nN_c + \tau]\cdot c_{li}^*[n] \\ &= \sum_{i=0}^{63}\sum_{j=0}^{63} C_{n_{kj}n_{li}}[\tau] \end{aligned}$$

$$C_{n_{kj}n_{li}}[\tau] \equiv \frac{1}{2N_l}\sum_{n=0}^{3} s_{n_{kj}}[nN_c + \tau]\cdot c_{n_{li}}^*[n] \tag{5}$$

Using the above, each C-matrix element requires 64(64) = 4096 complex adds, or 8192 operations to calculate. (Compare with 2048 operations for the direct method.)

Alternately, the calculations can be represented in terms of 4-chip real code segments and the corresponding waveforms. Hence all correlation calculations can be represented in terms of 16(16)(64) = 16K fundamental real correlations.

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: Calculation of C-matrix Elements Date: August 10, 2000

1. Introduction

The C-matrix elements are used to calculate the R-matrices, which are used by the MDF interference cancellation routine. Each C-matrix element can be calculated as a dot product between the *k*th user's waveform and the *l*th user's code stream, each offset by some multipath delay. For this method of calculation, each time a user's multipath profile changes all C-matrix elements associated with the changed profile must be recalculated. It is estimated that a user profile changes every 100 ms. This number, however, is based on very little data, and there is considerable risk that profiles may change more rapidly and compromise real-time operation. In addition, there is a large amount of overhead that must be performed before each dot product. In a recent benchmark the overhead consumed nearly all of the time allocated for the entire C-matrix update. Finally, if the C-matrix is calculated as described above then an entire processor must be allocated for this calculation.

In view of the above observations a better approach is to pre-calculate the code correlations up-front when a user is added to the system. This calculation is performed over all possible code offsets and the calculations are stored in a large array, approximately 21 Mbytes in size. We will henceforth refer to this large matrix as the Γ matrix. The C-matrix elements are updated when a profile changes by extracting the appropriate elements from the Γ matrix and performing minor calculations. Since the Γ matrix elements are calculated for all code offsets the FFT can be effectively used to speed up the calculations. Since all code offsets are pre-calculated, there is no risk associated with rapidly changing multipath profiles. Under normal operating conditions when the number of users accessing system is constant the resources which must be allocated to extracting the C-matrix elements are minimal, and so extra resources may be allocated to the R-matrix calculation.

Section 2 below outlines the calculation of the Γ matrix elements. It is shown that the Γ matrix elements are given in terms of a convolution. Section 3 shows how to calculate the Γ matrix elements using the FFT. Section 4 describes how the Γ-matrix elements might be

accessed from SDRAM. In section 5 various processing times are estimated, and a summary with conclusions is given in section 6.

2. C-matrix Elements Expressed in Terms of Code Correlations

The R-matrix elements are given in terms of the C-matrix elements as [1]

$$\hat{\rho}_{lk}[m']A_l A_k = \sum_{q=1}^{L}\sum_{q'=1}^{L}\mathrm{Re}\left\{\hat{a}_{lq}^{*}\hat{a}_{kq'}\cdot C_{lkqq'}[m']\right\}$$
$$C_{lkqq'}[m'] \equiv \frac{1}{2N_l}\sum_{n}s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^{*}[n] \tag{1}$$

where $C_{lkqq'}[m']$ is a five-dimensional matrix of code correlations. Both l and k range from 1 to K_v, where K_v is the number of virtual users. The indices q and q' range from 1 to L, the number of multipath components, which is assumed to be equal to 4. The symbol period offset m' ranges from −1 to 1. The total number of matrix elements to be calculated is then $N_C = 3(K_v L)^2 = 3(800)^2 = 1.92M$ complex elements, or 3.84 MB if each element is a byte. This number is cut in half, however, due to the symmetries [2]

$$C_{klq'q}[-m'] = \frac{N_l}{N_k}C_{lkqq'}^{*}[m'] \tag{2}$$

The memory requirement is then 1.92 MB.

Referring to Equation (1) it is evident that each element of $C_{lkqq'}[m']$ is a complex dot product between a code vector c_l and a waveform vector $s_{kqq'}$. The length of the code vector is 256. The waveform $s_k[t]$ is referred to as the signature waveform for the *kth* virtual user. This waveform is generated by passing the spread code sequence $c_k[n]$ through a pulse-shaping filter $g[t]$

$$s_k[t] = \sum_{p=0}^{N-1} g[t - pN_c]c_k[p] \tag{3}$$

where $N = 256$ and $g[t]$ is the raised-cosine pulse shape. Since $g[t]$ is a raised-cosine pulse as opposed to a root-raised-cosine pulse, the signature waveform $s_k[t]$ includes the effects of filtering by the matched chip filter. Note that for spreading factors less than *256* some of the chips $c_k[p]$ are zero. The length of the waveform vector is $L_g + 255N_c$, where L_g is the length of the raised-cosine pulse vector $g[t]$ and N_c is the number of samples per chip. The values for these parameters as currently implemented are $L_g = 48$ and $N_c = 4$. The length of the waveform vector is then 1068, but for the dot product it is accessed at a stride of $N_c = 4$, which gives effectively a length of 267.

The raised-cosine pulse vector $g[t]$ is defined to be non-zero from $t = -L_g/2 + 1{:}L_g/2$, with $g[0] = 1$. With this definition the waveform $s_k[t]$ is non-zero from $t = -L_g/2 + 1{:}\ L_g/2 + 255N_c$.

By combining Equations (1) and (3) the calculation of the C-matrix elements can be expressed directly in terms of the user code correlations. These correlations can be calculated up front and stored in SDRAM. The C-matrix elements expressed in terms of the code correlations $\Gamma_{lk}[m]$ are

$$\begin{aligned}
C_{lkqq'}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_p g[(n-p)N_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_k[p]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_m g[mN_c + \tau]\cdot c_k[n-m]\cdot c_l^*[n] \\
&= \sum_m g[mN_c + \tau]\cdot\frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m] \\
&= \sum_m g[mN_c + \tau]\cdot\Gamma_{lk}[m]
\end{aligned} \tag{4}$$

$$\Gamma_{lk}[m] \equiv \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m]$$
$$\tau \equiv m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}$$

Since the pulse shape vector $g[n]$ is of length L_g there are at most $2L_g/N_c = 24$ real macs to be performed to calculate each element $C_{lkqq'}[m']$. (The factor of 2 is because the code correlations $\Gamma_{lk}[m]$ are complex.) Given τ it is important to be able to efficiently calculate the range of values m for which $g[mN_c + \tau]$ is non-zero. The minimum value of m is given by $m_{min1}N_c + \tau = -L_g/2 + 1$. Now τ is given by $\tau = m'NN_c + \tau_{lq} - \tau_{kq'}$. If each τ value is decomposed $\tau_{lq} = n_{lq}N_c + p_{lq}$, then $m_{min1} = \text{ceil}[\,(-\tau - L_g/2 + 1)/N_c\,] = -m'N - n_{lq} + n_{kq'} - L_g/(2N_c) + \text{ceil}[\,(p_{kq'} - p_{lq} + 1)/N_c\,]$. Now $\text{ceil}[\,(p_{kq'} - p_{lq} + 1)/N_c\,]$ will be either 0 or 1. It is convenient to set this to 0. In order that we do not access values outside the allocation for $g[n]$ we must set $g[n] = 0.0$ for $n = -L_g/2 : -L_g/2 - (N_c - 1)$. Note that of the N_c^2 possible values for $\text{ceil}[\,(p_{kq'} - p_{lq} + 1)/N_c\,]$, all but one are 0. Hence we have

$$m_{\min 1} = -m'N - n_{lq} + n_{kq'} - L_g/(2N_c) \tag{5}$$

Note that L_g must be divisible by $2N_c$, and that $L_g/(2N_c)$ should be a system constant.

The maximum value of m is given by $m_{max1}N_c + \tau = L_g/2$. This gives $m_{max1} = \text{floor}[\,(-\tau + L_g/2)/N_c\,] = -m'N - n_{lq} + n_{kq'} + L_g/(2N_c) + \text{floor}[\,(p_{kq'} - p_{lq})/N_c\,]$. Now $\text{floor}[\,(p_{kq'} - p_{lq})/N_c\,]$ will be either -1 or 0. It is convenient to set this to 0. In order that we do not access values outside the allocation for $g[n]$ we must set $g[n] = 0.0$ for $n = -L_g/2 + 1 : L_g/2 + N_c$. Note that of the N_c^2 possible values for $\text{floor}[\,(p_{kq'} - p_{lq})/N_c\,]$, about half are 0. Hence we have

$$m_{\max 1} = -m'N - n_{lq} + n_{kq'} + L_g/(2N_c) \tag{6}$$

These values are quickly calculable.

The Γ matrix is calculated in the next section for all values m by exploiting the FFT. Notice that the calculation of the C-matrix elements requires only a small subset of the Γ matrix elements.

3. Using the FFT to Calculate the Γ-matrix Elements

In the previous section it was shown that the Γ-matrix elements can be represented as a convolution. This fact is here exploited to calculate the Γ-matrix elements using the FFT convolution theorem. From Equation (4) the Γ-matrix elements are

$$\Gamma_{lk}[m] \equiv \frac{1}{2N_l}\sum_{n=0}^{N-1} c_l^*[n]\cdot c_k[n-m] \tag{7}$$

where $N = 256$. Three streams are related by this equation. In order to apply the convolution theorem all three streams must be defined over the same time interval. The code streams $c_k[n]$ and $c_l[n]$ are non-zero from $n = 0{:}255$. These intervals are based on the maximum spreading factor. For higher data-rate users the intervals over which the streams are non-zero are reduced further. We are concerned here, however, with the intervals derived from the highest spreading factor since these will be the largest intervals and we wish to define a common interval for all streams. The common interval allows the FFTs to be reused for all user interactions.

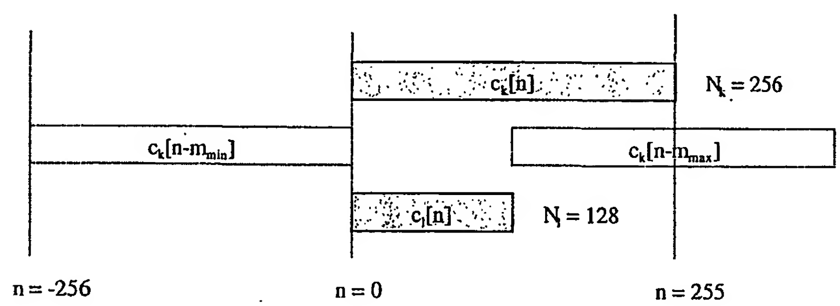


Figure 1. Interval for FFT calculation of the Γ matrix elements. Shown For the case where $N_k = 256$ and $N_l = 128$.

The range of values m for which $\Gamma_{lk}[m]$ is non-zero can be derived from the above intervals. The maximum value of m is limited by $n-m \geq 0$, which gives

$$255 - m_{\max} = 0 \;\;\Rightarrow\;\; m_{\max} = 255 \tag{8}$$

and the minimum value is limited by $n-m \leq 255$, which gives

$$0 - m_{\min} = 255 \;\;\Rightarrow\;\; m_{\min} = -255 \tag{9}$$

To achieve a common interval for all three streams we select the interval $m = -M/2{:}\,M/2-1$, $M = 512$. Where necessary the streams are zero-padded to fill up the interval.

Now, the DFT and IDFT of the streams are

$$C_l[r]=\sum_{n=-\frac{M}{2}}^{\frac{M}{2}-1} c_l[n]\cdot e^{-j2\pi nr/M}$$
$$c_l[n]=\frac{1}{M}\sum_{r=-\frac{M}{2}}^{\frac{M}{2}-1} C_l[r]\cdot e^{j2\pi nr/M} \tag{10}$$

which gives

$$\Gamma_{lk}[m]\equiv\frac{1}{2N_l}\sum_{n=-\frac{M}{2}}^{\frac{M}{2}-1} c_k[n-m]\cdot c_l^*[n]$$

$$=\frac{1}{2N_lM^2}\sum_{n=-\frac{M}{2}}^{\frac{M}{2}-1}\sum_{r=-\frac{M}{2}}^{\frac{M}{2}-1} C_k[r]\cdot e^{j2\pi(n-m)r/M}\sum_{r'=-\frac{M}{2}}^{\frac{M}{2}-1} C_l^*[r']\cdot e^{-j2\pi nr'/M} \tag{11}$$

$$=\frac{1}{2N_lM^2}\sum_{r=-\frac{M}{2}}^{\frac{M}{2}-1} C_k[r]\cdot e^{-j2\pi mr/M}\sum_{r'=-\frac{M}{2}}^{\frac{M}{2}-1} C_l^*[r']\sum_{n=-\frac{M}{2}}^{\frac{M}{2}-1} e^{j2\pi n(r-r')/M}$$

$$=\frac{1}{2N_lM}\sum_{r=-\frac{M}{2}}^{\frac{M}{2}-1} C_k[r]\cdot C_l^*[r]e^{-j2\pi mr/M}$$

Hence $\Gamma_{lk}[m]$ can be calculated using the FFTs. Notice that the FFT gives values for all m. From the analysis above we know that many of these values will be zero for high data rate users. To conserve memory we wish to store only the non-zero values. The values of m for which $\Gamma_{lk}[m]$ is non-zero can be determined analytically. This subject is treated in the next section where the storage and retrieval of the Γ–matrix elements is considered.

4. Storage and Retrieval of Γ-matrix Elements

In order to efficiently store the Γ–matrix elements we must determine which values are non-zero. For high data rate users certain elements $c_l[n]$ are zero, even within the interval $n = 0:N-1$, $N = 256$. These zero values reduce the interval over which $\Gamma_{lk}[m]$ is non-zero. In order to determine the interval for non-zero values consider

$$\Gamma_{lk}[m]\equiv\frac{1}{2N_l}\sum_{n=0}^{N-1} c_l^*[n]\cdot c_k[n-m] \tag{12}$$

Define index j_l for the lth virtual user such that $c_l[n]$ is non-zero only over the interval $n = j_l N_l : j_l N_l + N_l - 1$. Correspondingly, the vector $c_k[n]$ is non-zero only over the interval $n = j_k N_k : j_k N_k + N_k - 1$. Given these definitions $\Gamma_{lk}[m]$ can be rewritten as

$$\Gamma_{lk}[m] \equiv \frac{1}{2N_l} \sum_{n=0}^{N_l-1} c_l^*[n + j_l N_l] \cdot c_k[n + j_l N_l - m] \tag{13}$$

The minimum value of m for which $\Gamma_{lk}[m]$ is non-zero is

$$m_{\min 2} = -j_k N_k + j_l N_l - N_k + 1 \tag{14}$$

and the maximum value of m for which $\Gamma_{lk}[m]$ is non-zero is

$$m_{\max 2} = N_l - 1 - j_k N_k + j_l N_l \tag{15}$$

The total number of non-zero elements is then

$$\begin{aligned} m_{total} &\equiv m_{\max 2} - m_{\min 2} + 1 \\ &= N_l + N_k - 1 \end{aligned} \tag{16}$$

Table 1 below gives the number of bytes per l,k virtual-user pair based on 2 bytes per element – one byte for the real part and one byte for the imaginary part.

Table 1. Number of bytes per l,k virtual user pair based on 2 bytes per element.

	$N_k = 256$	128	64	32	16	8	4
$N_l = 256$	1022	766	638	574	542	526	518
128	766	510	382	318	286	270	262
64	638	382	254	190	158	142	134
32	574	318	190	126	94	78	70
16	542	286	158	94	62	46	38
8	526	270	142	78	46	30	22
4	518	262	134	70	38	22	14

Now we are in a position to determine the memory requirements for the Γ matrix for a given number of users at each spreading factor. Let there be K_q virtual users at spreading factor $N_q \equiv 2^{8-q}$, $q = 0{:}6$, where K_q is the qth element of the vector K. Note that some elements of K may be zero. Let Table 1 above be stored in matrix M with elements $M_{qq'}$. For example, $M_{00} = 1022$, and $M_{01} = 766$. The total memory required by the Γ matrix in bytes is then

$$\begin{aligned} M_{bytes} &= \sum_{q=0}^{6} \left\{ \frac{K_q(K_q+1)}{2} M_{qq} + \sum_{q'=q+1}^{6} K_q K_{q'} M_{qq'} \right\} \\ &= \frac{1}{2} \sum_{q=0}^{6} \left\{ K_q M_{qq} + \sum_{q'=0}^{6} K_q K_{q'} M_{qq'} \right\} \end{aligned} \tag{17}$$

For example, for 200 virtual users at spreading factor $N_0 = 256$ we have $K_q = 200\delta_{q0}$, which gives $M_{bytes} = ½K_0(K_0 + 1)M_{00} = 100(201)(1022) = 20.5$ MB.

For 10 384 Kbps users we have $K_q = K_0\delta_{q0} + K_6\delta_{q6}$ with $K_0 = 10$ and $K_6 = 640$. This gives $M_{bytes} = ½K_0(K_0 + 1)M_{00} + K_0K_6M_{06} + ½K_6(K_6 + 1)M_{66} = 5(11)(1022) + 10(640)(518) + 320(641)(14) = 6.2$ MB.

Now consider addressing, storing and accessing the Γ-matrix data. For each pair *(l,k), k >= l* we have 1 complex value $\Gamma_{lk}[m]$ value for each value of m, where m ranges from m_{min2} to m_{max2}, and the total number of non-zero elements is $m_{total} = m_{max2} - m_{min2} + 1$. Hence for each pair *(l,k), k >= l* we have $2m_{total}$ time-contiguous bytes. To access the data, create an array of structures:

```
struct {
        int m_min2;
        int m_max2;
        int m_total;
        char * Glk;
} G_info[N_VU_MAX][ N_VU_MAX];
```

The C-matrix data is then retrieved using something like:

```
m_min2 = G_info[l][k].m_min2
m_max2 = G_info[l][k].m_max2
N_g = L_g/N_c
N1 = m'*N – L_g/(2N_c)
for m' = 0:1
        for q = 0:L –1
                for q' = 0:L –1
                        τ = m'T + τ_lq – τ_kq'
                        m_min1 = N1 – n_lq + n_kq'
                        m_max1 = m_min1 + N_g
                        m_min = max[ m_min1 , m_min2 ]
                        m_max = min[ m_max1 , m_max2 ]
                        if m_max >= m_min
                                m_span = m_max – m_min + 1
                                sum1 = 0.0;
                                ptr1 = &G_info[l][k].Glk[m_min]
                                ptr2 = &g[ m_min * N_c + τ ]
                                while m_span > 0
                                        sum1 += ( *ptr1++ ) * ( *ptr2++ )
                                        m_span—
                                end
                                C[m'][l][k][q][q'] = sum1
                        end
                end
        end
end
```

5. Estimated Processing Times

The following processing times are estimated below:

- Calculate Γ-matrix elements
- Write to Γ-matrix elements to SDRAM
- Pack Γ-matrix elements in SDRAM
- Extract Γ-matrix elements/Form C-matrix from SDRAM
- Write C-matrix elements to L2 cache
- Pack C-matrix elements in L2 cache

Processing times are calculated for two cases of interest. The first case is where $K = 100$ users ($K_v = 200$ virtual users) are accessing the system and a voice user is added to the system. Not all of these users are active. The control channels are always active, but the data channels have activity factor AF = 0.4. The mean number of active virtual users is then $K + AF*K = 140$. The standard deviation is $\sigma = \sqrt{K \cdot AF \cdot (1-AF)} = 4.90$. With high probability, then, we have $K_v < 140 + 3\sigma < 155$ active users.

The second case is the worst case scenario. This occurs when a number of voice users are accessing the system and a single 384 Kbps data user is added. A single 384 Kbps data user adds interference equal to (.25 + 0.125*100)/(.25 + 0.400*1) ~= 20 voice users. Hence, the number of voice users accessing the system must be reduced to approximately $K = 100 - 20 = 80$ ($K_v = 160$). The 3σ number of active virtual users is then 80 + (0.125)80 +3(3.0) = 99 active virtual users. The reason this scenario is stressful is that when a single 384 Kbps data user is added to the system, $J + 1 = 64 + 1 = 65$ virtual users are added to the system.

Calculate Γ-matrix elements

The Γ-matrix elements can be calculated in one of two ways. The first is using the SAL zconvx to perform the direct convolution. The second is using the SAL fft_zipx to perform the calculation via the FFT. The first method is preferable when the vector lengths are small. SAL timing are given in Table 2. These timings are based on a 400 MHz PPC7400 with 160MHz, 2MB L2 cache. The data is assumed resident in L1 cache. The performance loss for data L2 cache resident is not severe.

Table 2. SAL timings and GFLOPS for zconvx function

M_{total}	N_l	Timing (µs)	GFLOPS
1024	4	19.33	1.70
1024	8	29.73	2.20
1024	16	50.55	2.59
1024	32	92.32	2.84
1024	64	176.53	2.97
1024	128	346.80	3.47

The time to perform a 512 complex FFT, with in-place calculation (fft_zipx), on a 400 MHz PPC7400 with 160MHz, 2MB L2 cache is 10.94 µs for data L1 resident. Prior to

performing the (final) FFT we must perform a complex vector multiply of length 512. The SAL timings for zvmulx are given in Table 3.

Table 3. SAL timings and GFLOPS for zvmulx function

Length	Location	Timing (μs)	GFLOPS
1024	L1	4.46	1.38
1024	L2	24.27	0.253
1024	DRAM	61.49	0.100

We will also be interested in the time to move data. Hence the SAL timings for zvmovx are given in Table 4.

Table 4. SAL timings for zvmovx function

Length	Location	Timing (μs)
1024	L1	1.20
1024	L2	15.34
1024	DRAM	30.05

Figure 2 shows the elements that must be calculated (in gray) when a physical user is added to the system. When a physical user is added to the system there are $1 + J$ virtual users added to the systems: that is, 1 control channel + $J = 256/SF$ data channels. The number K_v represents the number of virtual users that are using the system to begin with.

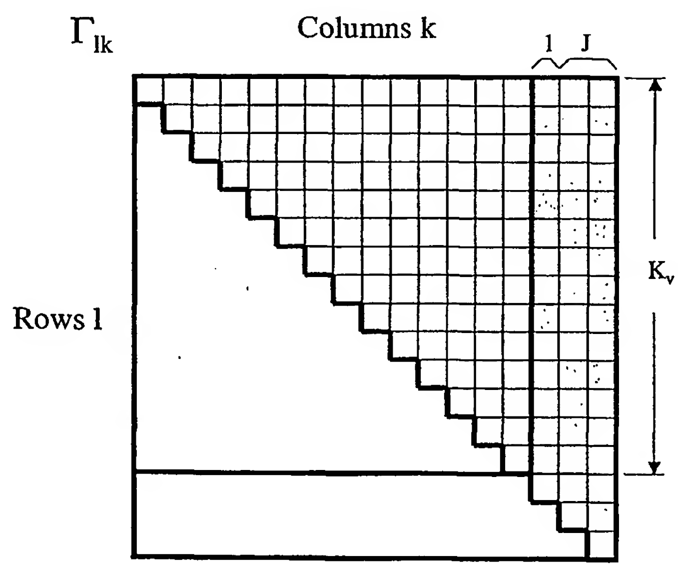


Figure 2. Elements that must be calculated (in gray) when a physical user is added to the system.

Hence there are $(K_v + 1)$ elements added due to the control channel, and $J(K_v + 1) + J(J + 1)/2$ elements added due to the data channels. The total number of elements added is then $(J + 1)[K_v + 1 + J/2]$.

Suppose that the FFT is used to perform the calculations. The total number of FFTs to perform is $(J + 1) + (J + 1)[K_v + 1 + J/2]$. The first term represents the FFTs to transform $c_k[n]$, and the second term represents the $(J + 1)[K_v + 1 + J/2]$ inverse FFTs of $\mathrm{FFT}\{c_k[n]\}^*\mathrm{FFT}\{c_l^*[n]\}$. The time to perform the complex 512 FFTs is 10.94 µs, whereas the time to perform the complex vector multiply and the complex 512 FFT is 24.27/2 + 10.94 = 23.08 µs.

For the first scenario there are $K_v = 200$ virtual users accessing the system and a voice user is added to the system ($J = 1$). The total time to add the voice user is then (1 + 1)(10.94 µs) + (1 + 1)[200 + 1 + 1/2](23.08 µs) = 9.3 ms.

For the second scenario there are $K_v = 160$ virtual users accessing the system and a 384 Kbps data user is added to the system ($J = 64$). The total time to add the 384 Kbps user is then (64 + 1)(10.94 µs) + (64 + 1)[160 + 1 + 64/2](23.08 µs) = 290 ms! This number is way too big and hence for high data-rate users, at least, the Γ-matrix elements must be calculated via convolutions.

The direct method to calculate the Γ-matrix elements is to use the SAL zconvx function to perform the convolution

$$\begin{aligned}\Gamma_{lk}[m] &\equiv \frac{1}{2N_l}\sum_{n=0}^{N_l-1} c_l^*[n+j_lN_l]\cdot c_k[n+j_lN_l-m] \\ &= \frac{1}{2N_l}\sum_{n=0}^{N_k-1} c_l^*[n+j_kN_k+m]\cdot c_k[n+j_kN_k]\end{aligned} \tag{18}$$

For each value of m there are $N_{min} = \min\{N_l, N_k\}$ complex macs (cmacs). Each cmac requires 8 flops, and there are $m_{total} = N_l + N_k - 1$ m-values to calculate. Hence the total number of flops is $8N_{min}(N_l + N_k - 1)$. For what follows we assume the convolution calculation is performed at 1.50 GOPs = 1500 ops/µs. The calculation time to perform the convolutions is presented in Table 5.

Table 5. Calculation time(µs) to perform the Γ-matrix convolutions.

	$N_k = 256$	128	64	32	16	8	4
$N_l = 256$	697.69	261.46	108.89	48.98	23.13	11.22	5.53
128	261.46	174.08	65.19	27.14	12.20	5.76	2.79
64	108.89	65.19	43.35	16.21	6.74	3.03	1.43
32	48.98	27.14	16.21	10.75	4.01	1.66	0.75
16	23.13	12.20	6.74	4.01	2.65	0.98	0.41
8	11.22	5.76	3.03	1.66	0.98	0.64	0.23
4	5.53	2.79	1.43	0.75	0.41	0.23	0.15

The shaded cells indicate times faster than the 23.08 µs FFT time. Equation 17 gives the size of the Γ-matrix in bytes. Similarly, the total time to calculate the Γ-matrix is

$$\begin{aligned} T_\Gamma(K) &= \sum_{q=0}^{6}\left\{\frac{K_q(K_q+1)}{2}T_{qq} + \sum_{q'=q+1}^{6}K_q K_{q'}T_{qq'}\right\} \\ &= \frac{1}{2}\sum_{q=0}^{6}\left\{K_q T_{qq} + \sum_{q'=0}^{6}K_q K_{q'}T_{qq'}\right\} \\ &= \frac{1}{2}\left[K\cdot diag(T) + K^T\cdot T\cdot K\right] \end{aligned} \tag{19}$$

where T_{qq} are the elements in Table 5. Now suppose $K' = K + \Delta$, where $\Delta_q = J_x\delta_{qx} + J_y\delta_{qy}$, where x and y are not equal. Then

$$\begin{aligned} \Delta T_\Gamma &\equiv T_\Gamma(K') - T_\Gamma(K) \\ &= \frac{1}{2}J_x(J_x+1)T_{xx} + \frac{1}{2}J_y(J_y+1)T_{yy} + J_x J_y T_{xy} + \sum_{q=0}^{6}K_q\left\{J_x T_{xq} + J_y T_{yq}\right\} \end{aligned} \tag{20}$$

For the first scenario there are $K_v = 200$ virtual users accessing the system and a voice user is added to the system ($J = 1$). Hence we have $K_q = K_v\delta_{q0}$ (SF = 256), $K_v = 200$, $J_x = J = 2$ and $J_y = 0$. The total time is then

$½J(J+1)T_{00} + JK_vT_{00}$ = (0.5)(2)(3)(0.70 ms) + (2)(200)(0.70 ms) = 283 ms

This number is way too big and hence for voice users, at least, the Γ-matrix elements must be calculated via FFTs.

For the second scenario there are $K_v = 160$ virtual users accessing the system and a 384 Kbps data user is added to the system ($J = 64$). Hence we have $K_q = K_v\delta_{q0}$ (SF = 256), $K_v = 160$, $J_x = 1$ (control) and $J_y = J = 64$ (data). The total time is then

$(K_v + 1)T_{00} + J(K_v + 1)T_{06} + (J + 1)(J/2)T_{66}$
= (161)(697.7 μs) + (64)(161)(5.53 μs) + (65)(32)(0.15 μs) =
112.33 ms + 56.98 ms + 0.31 ms = 169.62 ms

Since T_{00} = 697.7 μs is so large, these calculations should be performed using the FFT, which costs 23.08 μs per convolution. We also have 1 FFTs to compute FFT{$c_k^*[n]$}) for the single control channel. This costs an additional 10.94 μs. The total time, then, to add the 384 Kbps user is

10.94 μs + (161)(23.08) μs + (64)(161)(5.53) μs + (65)(32)(0.15) μs =
= 61.02 ms

Write to Γ-matrix elements to SDRAM

The numbers in Table 1 represent the $2m_{total}$ bytes per Γ-matrix element. Recall that the size of the Γ-matrix in bytes from Equation 17 is

$$\begin{aligned} \dot{M}_b(K) &= \sum_{q=0}^{6}\left\{\frac{K_q(K_q+1)}{2}M_{qq} + \sum_{q'=q+1}^{6} K_q K_{q'} M_{qq'}\right\} \\ &= \frac{1}{2}\sum_{q=0}^{6}\left\{K_q M_{qq} + \sum_{q'=0}^{6} K_q K_{q'} M_{qq'}\right\} \\ &= \frac{1}{2}\left[K \cdot diag(M) + K^T \cdot M \cdot K\right] \end{aligned} \quad (21)$$

Now suppose $K' = K + \Delta$, where $\Delta_q = J_x\delta_{qx} + J_y\delta_{qy}$, where x and y are not equal. Then

$$\begin{aligned} \Delta M_b &\equiv M_b(K') - M_b(K) \\ &= \frac{1}{2}J_x(J_x+1)M_{xx} + \frac{1}{2}J_y(J_y+1)M_{yy} + J_x J_y M_{xy} \\ &+ \sum_{q=0}^{6} K_q\left\{J_x M_{xq} + J_y M_{yq}\right\} \end{aligned} \quad (22)$$

Consider the first scenario where $K_q = 200\delta_{q0}$ (SF = 256) and that a single voice user is added to the system: $J_x = 2$ (data plus control), and $J_y = 0$. The total number of bytes is then 0.5(2)(3)(1022) + 200(2)(1022) = 0.412 MB. The SDRAM write speed is 133MHz*8 bytes * 0.5 = 532 MB/s. The time to write to SDRAM is then 0.774 ms.

Now for the second scenario $K_q = 160\delta_{q0}$ (SF = 256), and that a single 384 Kbps (SF = 4) user is added to the system: $J_x = 1$ (control) and $J_y = 64$ (data). The total number of bytes is then 0.5(1)(2)(1022) + 0.5(64)(65)(14) + 160{1(1022) + 64(518)} = 5.498 MB. The SDRAM write speed is 133MHz*8 bytes * 0.5 = 532 MB/s. The time to write to SDRAM is then 10.33 ms.

Pack Γ-matrix elements in SDRAM

The maximum total size of the Γ-matrix is 20.5 MB. Suppose that in order to pack the matrix every element must be moved. This is the worst case. The SDRAM speed is 133MHz*8 bytes * 0.5 = 532 MB/s. The move time is then 2(20.5 MB)/(532 MB/s) = 77.1 ms. If the Γ-matrix is divided over three processors this time is reduced by a factor of 3. The packing can be done incrementally, so there is no strict time limit.

Extract Γ-matrix elements/Form C-matrix from SDRAM

Recall that the C-matrix data is retrieved using something like:

```
m_min2 = G_info[l][k].m_min2
m_max2 = G_info[l][k].m_max2
N_g = L_g/N_c
N1 = m'*N - L_g/(2N_c)
for m' = 0:1
        for q = 0:L -1
                for q' = 0:L -1
                        τ = m'T + τ_lq - τ_kq'
                        m_min1 = N1 - n_lq + n_kq'
                        m_max1 = m_min1 + N_g
                        m_min = max[ m_min1 , m_min2 ]
                        m_max = min[ m_max1 , m_max2 ]
                        if m_max >= m_min
                                m_span = m_max - m_min + 1
                                sum1 = 0.0;
                                ptr1 = &G_info[l][k].Glk[m_min]
                                ptr2 = &g[ m_min * N_c + τ ]
                                while m_span > 0
                                        sum1 += ( *ptr1++ ) * ( *ptr2++ )
                                        m_span--
                                end
                                C[m'][l][k][q][q'] = sum1
                        end
                end
        end
end
```

<u>Time to extract elements when a new user is added to the system</u>

We calculated above the time to calculate the Γ-matrix elements when a new user is added to the system. Here we consider the time to extract the corresponding C-matrix elements.

Notice that *Glk[m]* are accessed from SDRAM. Values will almost certainly *not* be in either L1 or L2 cache. For a given *(l,k)* pair, however, the spread in τ will for most cases be less than 8 μs (i.e for a 4 μs delay spread), which equates to (8 μs)(4 chips/μs)(2 bytes/chip) = 64 bytes, or 2 cache lines. Since data must be read in for two values of *m'* a total of 4 cache lines must be read. This will require 16 clocks, or about 16/133 = 0.12 μs. However, measured results for zvmovx indicate that accesses to SDRAM are performed at about 50% efficiency so that the required time is about 0.24 μs.

Now suppose, for example, user $l = x$ is added to the system. We must fetch the elements *C[m'][x][k][q][q']* for all *m', k, q* and *q'*. As indicated above, all the *m', q* and *q'* values will be contained typically in 4 cache lines. Hence if there are K_v virtual users we must read in $4K_v$ cache lines, or $32K_v$ clocks, where we have doubled the clocks to account for the 50%

efficiency. In general $J + 1$ virtual users are added to the system at a time. This will require $32K_v(J + 1)$ clocks.

For the first case where we have 155 active virtual users and a new voice user is added to the system, the time required to read in the C-matrix elements will be 32(155)(1 + 1) clocks/(133 clocks/μs) = 74.6 μs. The industry standard hold time t_h for a voice call is 140 s. The average rate λ of users added to the system can be determined from $\lambda t_h = K$, where K is the average number of users using the system. For $K = 100$ users we have λ = 100/140 s = 1 users added per 1.4 s.

For the case where we have 99 active virtual users and a 384 Kbps user is added to the system, the time required to read in the C-matrix elements will be 32(99)(64 + 1) clocks/(133 clocks/μs) = 1.55 ms. However data users presumably will be added to the system more infrequently than voice users.

Time to extract elements when τ_{xy} changes

Now suppose, for example, user $l = x$ lag $q = y$ changes. Then we must fetch the elements *C[m'][x][k][y][q']* for all m', k and q'. All the q' values will be contained typically in 1 cache line. Hence we must read in $2(K_v)(1) = 2K_v$ cache lines, or $16K_v$ clocks, where we have doubled the clocks to account for the 50% efficiency. In general, when a lag changes there are $J + 1$ virtual users for which the C-matrix elements must be updated. This will require $16K_v(J + 1)$ clocks.

For the first case where we have 155 active virtual users and a voice user's profile (one lag) changes, the time required to read in the C-matrix elements will be 16(155)(1 + 1) clocks/(133 clocks/μs) = 37.3 μs. Recall that for high mobility users such changes should occur at a rate of about 1 per 100 ms per physical user. This equates to about once per 1.33 ms processing interval if there are 100 physical users so that approximately 37.3 μs will be required every 1.33 ms.

For the case where we have 99 virtual users and a 384 Kbps data user's profile (one lag) changes, the time required to read in the C-matrix elements will be 16(99)(64 + 1) clocks/(133 clocks/μs) = 0.774 ms. However data users will have lower mobility and hence such changes should occur infrequently.

Write C-matrix elements to L2 cache

Time to write elements when a new user is added to the system

Consider again the case where user $l = x$ is added to the system. We must write elements *C[m'][x][k][q][q']* for all m', k, q and q'. If there are K_v active virtual users we must write $4K_vL^2$ bytes, where we have doubled the bytes since the elements are complex. In general $J + 1$ virtual users are added to the system at a time. This will require $4K_vL^2(J + 1)$ bytes to be written to L2 cache.

For the first case where we have 155 active virtual users and a new voice user is added to the system, the time required to write the C-matrix elements will be 4(155)(16)(1 + 1) bytes/(2128 bytes/μs) = 9.3 μs.

For the second case where we have 99 active virtual users and a 384 Kbps user is added to the system, the time required to write the C-matrix elements will be 4(99)(16)(64 + 1) bytes/(2128 bytes/μs) = 193.5 μs. Recall, however, that data users presumably will be added to the system more infrequently than voice users.

Time to extract elements when τ_{xy} changes

Now suppose, for example, user $l = x$ lag $q = y$ changes. We must write elements *C[m'][x][k][q][q']* for all m', k and q'. If there are K_v active virtual users we must write $4K_vL$ bytes, where we have doubled the bytes since the elements are complex. In general $J + 1$ virtual users are added to the system at a time. This will require $4K_vL(J + 1)$ bytes to be written to L2 cache.

For the first case where we have 155 active virtual users and a voice user's profile (one lag) changes, the time required to write the C-matrix elements will be 4(155)(4)(1 + 1) bytes/(2128 bytes/μs) = 2.33 μs.

For the second case where we have 99 active virtual users and a 384 Kbps data user's profile (one lag) changes, the time required to write the C-matrix elements will be 4(99)(4)(64 + 1)bytes/(2128 bytes/μs) = 48.4 μs. However data users will have lower mobility and hence such changes should occur infrequently.

Pack C-matrix elements in L2 cache

The C-matrix elements will need to be packed in memory every time a new user is added to or deleted from the system and every time a new user becomes active or inactive. The size of the C-matrix is $2(3/2)(K_vL)^2 = 3(K_vL)^2$ bytes, however, divided over three processors this becomes $(K_vL)^2$ bytes per processor. Assume that the entire matrix must be moved. The move is within L2 cache. Hence the total move time is $2(K_vL)^2$ bytes/(2128 bytes/μs), where the factor of 2 accounts for read and write.

For the first case where we have 155 active virtual users the time required to move the C-matrix elements will be $2(155*4)^2$ bytes/(2128 bytes/μs) = 0.361 ms.

For the first case where we have 99 active virtual users the time required to move the C-matrix elements will be $2(99*4)^2$ bytes/(2128 bytes/μs) = 0.147 ms.

These events will occur typically once every 10 ms, that is, once per frame.

6. Summary and Conclusions

In summary, we have determined

- The Γ-matrix will require approximately 20.5 MB of SDRAM
- To efficiently calculate the Γ-matrix elements will require both direct convolution and FFT calculations
- To pack the Γ matrix in SDRAM will require approximately 77.1 ms

The following processing times are estimated:

Estimated Processing Times	Case 1 (voice user added)	Case 2 (384 Kbps user added)
Calculate Γ-matrix elements	9.3 ms	61.0 ms
Write Γ-matrix elements to SDRAM	0.77 ms	10.3 ms
Extract C-matrix elements when		
New user added	75 μs	1.6 ms
Multipath profile changes	37 μs	0.77 ms
Write C-matrix elements to L2 when		
New user added	9.3 μs	194 μs
Multipath profile changes	2.3 μs	48 μs
Pack C-matrix elements in L2 cache	361 μs	147 μs

These times are based on a single but devoted G4 allocated to perform the calculations.

References

[1] J. H. Oates, "MUD Algorithms," Mercury Wireless Communications Group Report, April 25, 2000.

[2] J. H. Oates, "R-matrix GOPS," Mercury Wireless Communications Group Report, June 21, 2000.

Computer Systems, Inc.
MERCURY
199 Riverneck Road
Chelmsford, MA 01824-2820
(978) 256-1300 • Fax (978) 256-3599
http://www.mc.com

Report

To: Wireless Communications Group

From: J. H. Oates

Subject: Hardware Calculation of Γ-matrix Elements Date: November 13, 2000

The C-matrix elements can be represented in terms of the underlying code correlations using

$$\begin{aligned}
C_{lkqq'}[m'] &\equiv \frac{1}{2N_l}\sum_n s_k[nN_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_p g[(n-p)N_c + m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}]\cdot c_k[p]\cdot c_l^*[n] \\
&= \frac{1}{2N_l}\sum_n\sum_m g[mN_c + \tau]\cdot c_k[n-m]\cdot c_l^*[n] \\
&= \sum_m g[mN_c + \tau]\cdot \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m] \\
&= \sum_m g[mN_c + \tau]\cdot \Gamma_{lk}[m]
\end{aligned} \tag{1}$$

$$\begin{aligned}
\Gamma_{lk}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m] \\
\tau &\equiv m'T + \hat{\tau}_{lq} - \hat{\tau}_{kq'}
\end{aligned}$$

The Γ-matrix represents the correlation between the complex user codes. The complex code for user l is assumed to be infinite in length, but with only N_l non-zero values. The non-zero values are constrained to be $\pm 1 \pm j$. The Γ-matrix can represented in terms of the real and imaginary parts of the complex user codes becomes

$$\begin{aligned}\Gamma_{lk}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^*[n]\cdot c_k[n-m] \\ &= \frac{1}{2N_l}\sum_n \left\{c_l^R[n]-jc_l^I[n]\right\}\cdot\left\{c_k^R[n-m]+jc_k^I[n-m]\right\} \\ &= \frac{1}{2N_l}\sum_n \left\{c_l^R[n]\cdot c_k^R[n-m]+c_l^I[n]\cdot c_k^I[n-m]\right. \\ &\qquad\qquad \left.+\,jc_l^R[n]\cdot c_k^I[n-m]-jc_l^I[n]\cdot c_k^R[n-m]\right\} \\ &= \Gamma_{lk}^{RR}[m]+\Gamma_{lk}^{II}[m]+j\left\{\Gamma_{lk}^{RI}[m]-\Gamma_{lk}^{IR}[m]\right\}\end{aligned} \tag{2}$$

where

$$\begin{aligned}\Gamma_{lk}^{RR}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^R[n]\cdot c_k^R[n-m] \\ \Gamma_{lk}^{II}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^I[n]\cdot c_k^I[n-m] \\ \Gamma_{lk}^{RI}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^R[n]\cdot c_k^I[n-m] \\ \Gamma_{lk}^{IR}[m] &\equiv \frac{1}{2N_l}\sum_n c_l^I[n]\cdot c_k^R[n-m]\end{aligned} \tag{3}$$

Consider any one of the above real correlations, denoted

$$\Gamma_{lk}^{XY}[m] \equiv \frac{1}{2N_l}\sum_n c_l^X[n]\cdot c_k^Y[n-m] \tag{4}$$

where X and Y can be either R or I. Since the elements of the codes are now constrained to be ± 1 or 0, we can define

$$c_l^X[n] = \left(1-2\gamma_l^X[n]\right)\cdot m_l^X[n] \tag{5}$$

where $\gamma_l^X[n]$ and $m_l^X[n]$ are both either zero or one. The sequence $m_l^X[n]$ is a mask used to account for values of $c_l^X[n]$ that are zero. With these definitions Equation (4) becomes

$$
\begin{aligned}
\Gamma_{lk}^{XY}[m] &\equiv \frac{1}{2N_l}\sum_n \left(1-2\gamma_l^X[n]\right)\cdot m_l^X[n]\cdot\left(1-2\gamma_k^Y[n-m]\right)\cdot m_k^Y[n-m] \\
&= \frac{1}{2N_l}\sum_n \left(1-2\gamma_l^X[n]\right)\cdot\left(1-2\gamma_k^Y[n-m]\right)\cdot m_l^X[n]\cdot m_k^Y[n-m] \\
&= \frac{1}{2N_l}\sum_n \left[1-2\left(\gamma_l^X[n]\oplus\gamma_k^Y[n-m]\right)\right]\cdot m_l^X[n]\cdot m_k^Y[n-m] \\
&= \frac{1}{2N_l}\left\{\sum_n \cdot m_l^X[n]\cdot m_k^Y[n-m] \right. \\
&\qquad\qquad \left. -2\sum_n\left(\gamma_l^X[n]\oplus\gamma_k^Y[n-m]\right)\cdot m_l^X[n]\cdot m_k^Y[n-m]\right\} \qquad (6) \\
&= \frac{1}{2N_l}\left\{M_{lk}^{XY}[m]-2N_{lk}^{XY}[m]\right\}
\end{aligned}
$$

$$
\begin{aligned}
M_{lk}^{XY}[m] &\equiv \sum_n \cdot m_l^X[n]\cdot m_k^Y[n-m] \\
N_{lk}^{XY}[m] &\equiv \sum_n\left(\gamma_l^X[n]\oplus\gamma_k^Y[n-m]\right)\cdot m_l^X[n]\cdot m_k^Y[n-m]
\end{aligned}
$$

where ⊕ indicates modulo-2 addition (or logical XOR).

The hardware to perform these operations is shown in Figures 1 – 3. Figure 1 shows the initial register configuration after loading code and mask sequences. The boolean functions are shown in Figure 2, and Figure 3 shows the register configuration after a number of shifts.

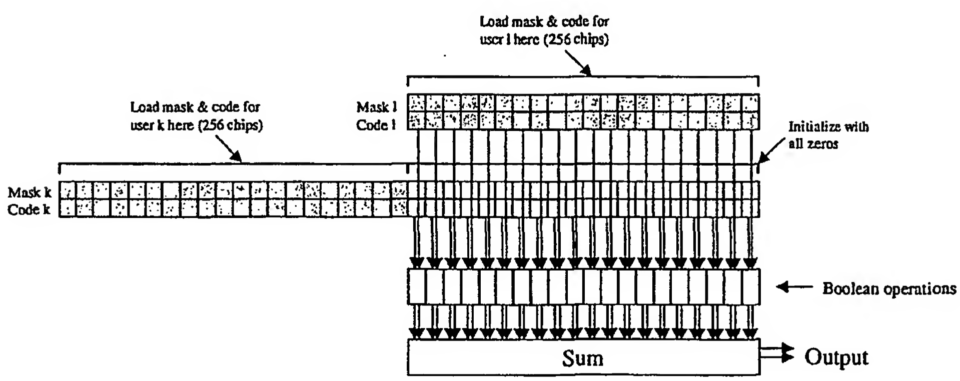


Figure 1. Initial register configuration after loading code and mask sequences.

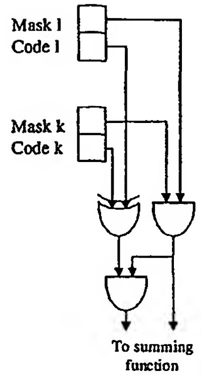


Figure 2. Boolean functions.

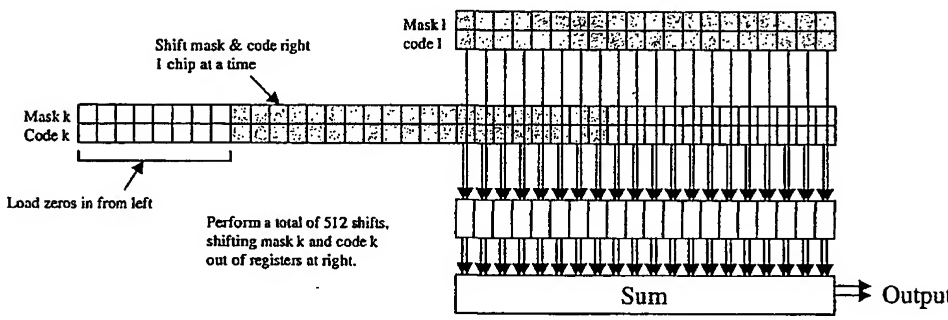
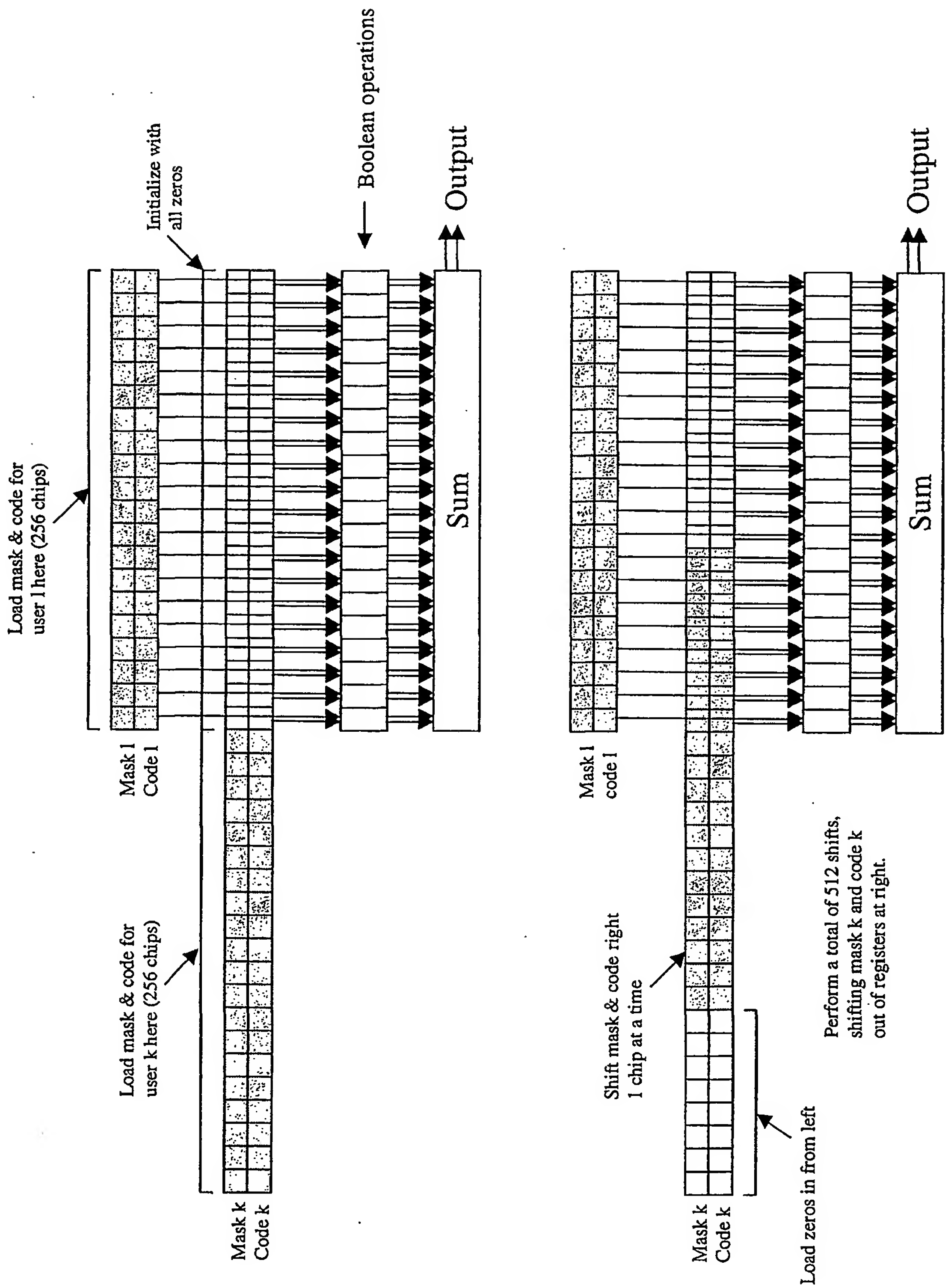
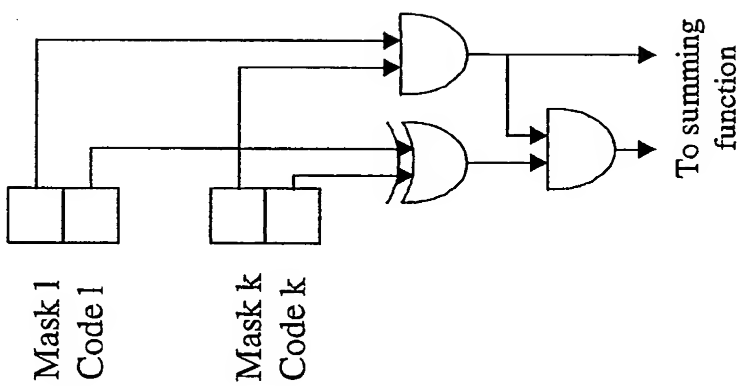


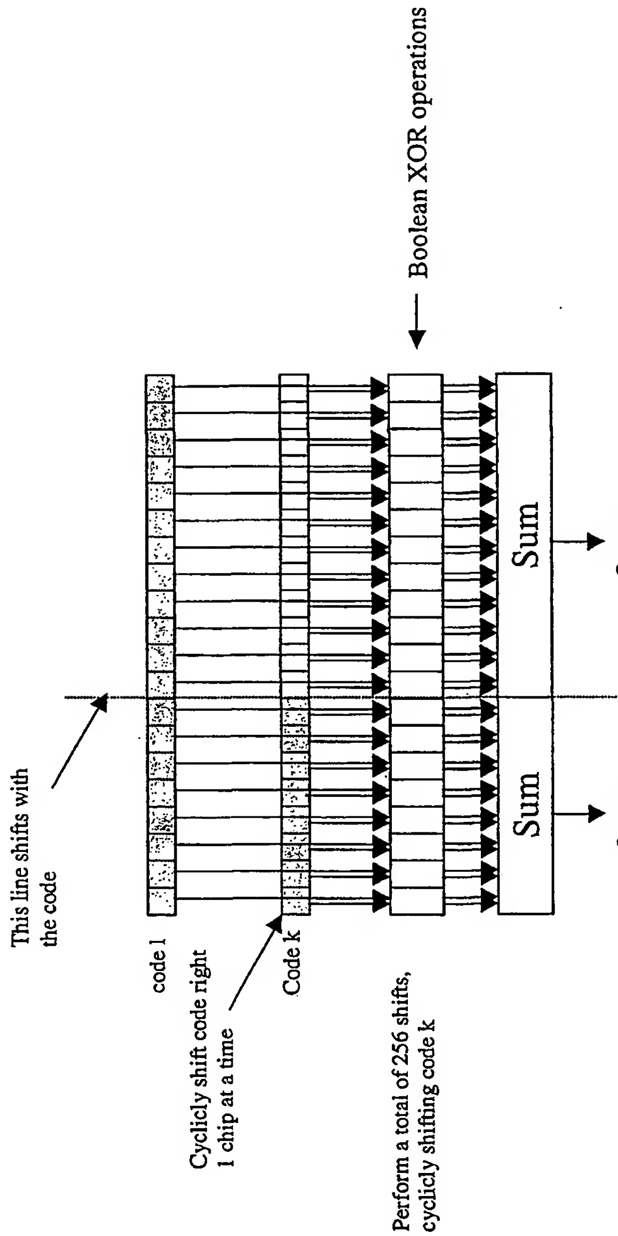
Figure 3. Register configuration after a number of shifts.

The above hardware calculates the functions $M_{lk}^{XY}[m]$ and $N_{lk}^{XY}[m]$. The remaining calculations to form $\Gamma_{lk}^{XY}[m]$ and subsequently $\Gamma_{lk}[m]$ can be performed in software. Note that the four functions $\Gamma_{lk}^{XY}[m]$ corrsponding to *X, Y* = *R, I* which are components of $\Gamma_{lk}[m]$ can be calculated in parallel. For $K_v = 200$ virtual users, and assuming that 10% of all (*l, k*) pairs must be calculated in 2 ms, then for real-time operation we must calculate $0.10(200)^2 = 4000$ $\Gamma_{lk}[m]$ elements (all shifts) in 2 ms, or about 2M elements (all shifts) per second. For $K_v = 128$ virtual users the requirement drops to 0.8192M elements (all shifts) per second.

In what has been presented the $\Gamma_{lk}[m]$ elements are calculated for all 512 shifts. Not all of these shifts are needed, so it is possible to reduce the number of calculations per $\Gamma_{lk}[m]$ elements. The cost is increased design complexity.


Load mask & code for
user 1 here (256 chips)
Initialize with
all zeros
Boolean operations
Output
Sum
Mask 1
Code 1
Load mask & code for
user k here (256 chips)
Mask k
Code k
Output
Sum
Mask 1
code 1
Perform a total of 512 shifts,
shifting mask k and code k
out of registers at right.
Shift mask & code right
1 chip at a time
Load zeros in from left
Mask k
Code k


Mask 1
Code 1
Mask k
Code k
To summing function



Two outputs representing code correlation at offsets separated by 256 chips are produced every clock cycle. This idea can be extended to handle virtual-user code correlations, in which case many outputs are produced every clock cycle.

Hardware vs. Software Calculation of G-matrix Elements

- Calculation of G-matrix elements
  - Requires performing XOR, bit-sum, bit-masking and bit-shifting operations on 256-bit registers
  - Approximately one million elements must be calculated every second
- Problems with using the AltiVec
  - The AltiVec solution is approximately 40 times slower than the FPGA solution because:
    - The AltiVec does not have a bit-sum instruction
    - Two 128-bit AltiVec registers are required to represent a 256 bit register
  - The PPC/AltiVec processor draws from 8 to 10 Watts
- Advantages of an FPGA implementation
  - XOR, bit-sum, bit-masking and bit-shifting operations are ideally suited for FPGA implementation
  - The G-matrix calculations are fundamental to MUD operation and will be identical for any variations in MUD algorithm
  - The FPGA implementation will run real-time with a single FPGA
  - The FPGA draws only 2 to 3 Watts
  - The slower FPGA clock speed can be counterbalanced by implementing multiple calculation functions in parallel

Makefile 2/23/2001

```
#
.SUFFIXES: .a .c .mac .o .S

ARCH = ppc7400
MUDLIB = mudlib.a

###CFLAGS = -Ot -t ${ARCH} -I. -DCOMPILE_C
CFLAGS = -Ot -t ${ARCH} -I.
ASFLAGS = -t ${ARCH} -DBUILD_MAX -I.

#
# Make object files
#
.c.o:
	ccmc ${CFLAGS} -o $*.o -c $*.c

#
#	Make ASM
#
.mac.o:
	rm -f $*.S
	cp $*.mac $*.S
	ccmc ${ASFLAGS} -o $*.o -c $*.S
	rm -f $*.S

OBJS = \
	get sizes.o \
	get sizes v.o \
	reformat corr.o \
	rmats.o \
	reformat_r.o \
	mpic.o \
	gen x row.o \
	gen r sums.o \
	gen r sums2.o \
	gen r matrices.o \
	mtrans32 8bit.o \
	mtriangle 8bit.o \
	dotpr3 8bit.o \
	dotpr6 8bit.o \
	dotpr9 8bit.o \
	sve3 8bit.o \
	fixed cdotpr.o \
	zdotpr4 vmx.o \
	zdotpr_vmx.o

${MUDLIB}: Makefile ${OBJS}
	armc -c $@ ${OBJS}

#
# Cleanup
#
clean:
	rm -f ${OBJS} *.S ${MUDLIB}

get sizes.o: mudlib.h get_sizes.c
reformat_corr.o: mudlib.h reformat_corr.c
rmats.o: mudlib.h rmats.c \
			gen x row.mac gen r_sums.mac gen_r_sums2.mac
			gen r matrices.mac
reformat r.o: mudlib.h reformat_r.c
mpic.o: mudlib.h mpic.c \
			dotpr3 8bit.mac dotpr6_8bit.mac dotpr9_8bit.mac
			sve3_8bit.mac

dotpr3 8bit.o: dotpr3 8bit.mac salppc.inc
dotpr6_8bit.o: dotpr6_8bit.mac salppc.inc
```

Makefile 2/23/2001

```
dotpr9 8bit.o: dotpr9 8bit.mac salppc.inc
sve3 8bit.o: sve3 8bit.mac salppc.inc
fixed cdotpr.o: zdotpr4 vmx.mac salppc.inc
zdotpr4_vmx.o: zdotpr4_vmx.mac zdotpr4_vmx.k salppc.inc
```

rmats.c 2/23/2001

```
#include "mudlib.h"

#define  DO CALC STATS  0
#define  DO TRUNCATE  1
#define  DO SATURATE  1
#define  DO_SQUELCH  0

#define  SQUELCH THRESH  1.0
#define  TRUNCATE_BIAS  0.0

#if DO TRUNCATE
#define SATURATE_THRESH  (128.0 + TRUNCATE_BIAS)
#else
#define SATURATE_THRESH  127.5
#endif

#define SATURATE( f ) \
  { \
    if ( (f) >= SATURATE THRESH ) f = (SATURATE THRESH - 1.0); \
    else if ( (f) < -SATURATE_THRESH ) f = -SATURATE_THRESH; \
  }

#if DO_TRUNCATE
#if 0
#define  BF8_FIX( f )   ((BF8) (FABS(f) <= TRUNCATE BIAS) ? 0 : \
                        (((f) > 0.0) ? ((f) - TRUNCATE BIAS) : \
                                       ((f) + TRUNCATE_BIAS)))
#define  BF8_FIX( f )   ((BF8)(f))
#else
#define  BF8_FIX( f )   ((BF8)(((((f) < 0.0)) && ((f) == (float)((int)(f)))) ?
\
                        ((f) + 1.0) : (f)))
#endif

#else
#define  BF8_FIX( f )   ((BF8)(((f) >= 0.0) ? ((f)+0.5) : ((f)-0.5)))
#endif

#define  UPDATE MAX( f, max ) \
  if ( FABS( f ) > max ) max = FABS( f );

#define uchar  unsigned char
#define ushort  unsigned short
#define ulong  unsigned long

#if DO_CALC STATS
static float  max_R_value;
#endif

void  gen X row (
        COMPLEX BF16  *mpath1 bf,
        COMPLEX BF16  *mpath2_bf,
        COMPLEX BF16  *X_bf,
        int  phys index,
        int  tot_phys_users
      );

void  gen R sums (
       COMPLEX BF16  *X bf,
       COMPLEX BF8  *corr_bf,
       uchar  *ptov map,
       BF32  *R sums,
       int  num_phys_users
     );

void  gen_R_sums2 (
```

```
        COMPLEX BF16  *X bf,
        COMPLEX BF8  *corra bf,
        COMPLEX BF8  *corrb_bf,
        uchar  *ptov map,
        BF32  *R sumsa,
        BF32  *R sumsb,
        int  num_phys_users
     );

void  gen R matrices (
         BF32  *R sums,
         float  *bf scalep,
         float  *inv scalep,
         float  *scalep,
         BF8  *no scale row bf,
         BF8  *scale row bf,
         int  num_virt_users
      );

void  mudlib gen R (
         COMPLEX BF16  *mpath1 bf,
         COMPLEX BF16  *mpath2 bf,
         COMPLEX BF8  *corr 0 bf,     /* adjusted for starting physical user */
         COMPLEX BF8  *corr_1_bf,     /* adjusted for starting physical user */
         uchar  *ptov map,            /* no more than 256 virts. per phys */
         float  *bf scalep,           /* scalar: always a power of 2 */
         float  *inv scalep,          /* start at 0'th physical user */
         float  *scalep,              /* start at 0'th physical user */
         char  *L1 cachep,            /* temp: 32K bytes, 32-byte aligned */
         BF8  *R0 upper bf,
         BF8  *R0 lower bf,
         BF8  *R1 trans_bf,
         BF8  *R1m bf,
         int  tot phys users,
         int  tot virt users,
         int  start phys user,        /* zero-based starting row (inclusive) */
         int  start virt user,        /* relative to start phys user */
         int  end phys user,          /* zero-based ending row (inclusive) */
         int  end_virt_user           /* relative to end_phys_user */
      )
{
  COMPLEX BF16  *X bf;
  BF32  *R sums0, *R sums1;
  uchar  *R0_ptov_map;

  int  bump, byte offset, i, iv, last virt user;
  int  R0_align, R0_skipped_virt_users, R0_tcols, R0_virt_users, R1_tcols;

#if DO CALC STATS
  max R_value = 0.0;
#endif

  X_bf = (COMPLEX_BF16 *)L1_cachep;

  byte offset = tot phys users * NUM FINGERS SQUARED * sizeof(COMPLEX BF16);
  R_sums0 = (BF32 *)(((ulong)X bf + byte_offset + R_MATRIX_ALIGN_MASK) &
            ~R_MATRIX_ALIGN_MASK);

  byte offset = tot virt users * sizeof(BF32);
  R_sums1 = (BF32 *)(((ulong)R sums0 + byte_offset + R_MATRIX_ALIGN_MASK) &
            ~R_MATRIX_ALIGN_MASK);

  R0_ptov_map = (uchar *)(((ulong)R sums1 + byte offset +
                   R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK);

  R1_tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;
```

```
  R0_virt_users = 0;
  for ( i = start_phys_user; i < tot_phys_users; i++ ) {
    R0_virt_users += (int)ptov_map[i];
    R0_ptov_map[i] = ptov_map[i];
  }

  R0_ptov_map[start_phys_user] -= start_virt_user;
  R0_skipped_virt_users = tot_virt_users - R0_virt_users + start_virt_user;
  R0_virt_users -= (start_virt_user + 1);

  --inv_scalep;            /* predecrement to allow for common indexing */

  for ( i = start_phys_user; i <= end_phys_user; i++ ) {

    gen_X_row (
      mpath1_bf,
      mpath2_bf,
      X_bf,
      i,
      tot_phys_users
    );

    --R0_ptov_map[i];                       /* excludes R0 diagonal */

    last_virt_user = (i < end_phys_user) ? ((int)ptov_map[i] - 1) :
                                            end_virt_user;

    for ( iv = start_virt_user; (iv + 1) <= last_virt_user; iv += 2 ) {

      gen_R_sums2 (
        X_bf + (i * NUM_FINGERS_SQUARED),
        corr_0_bf,
        corr_0_bf + ((R0_virt_users - 1) * NUM_FINGERS_SQUARED),
        R0_ptov_map + i,
        R_sums0 + (R0_skipped_virt_users + 1),
        R_sums1 + (R0_skipped_virt_users + 1),
        tot_phys_users - i
      );

      R0_tcols = R1_tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);
      R0_align = (R0_skipped_virt_users & R_MATRIX_ALIGN_MASK) + 1;

      gen_R_matrices (
        R_sums0 + (R0_skipped_virt_users + 1),
        bf_scalep,
        inv_scalep + (R0_skipped_virt_users + 1),
        scalep + (R0_skipped_virt_users + 1),
        R0_lower_bf + R0_align,
        R0_upper_bf + R0_align,
        R0_virt_users
      );

      R0_upper_bf[ R0_align - 1 ] = 0;  /* zero diagonal element */

      R0_lower_bf += R0_tcols;
      R0_upper_bf += R0_tcols;

      R0_tcols = R1_tcols - ((R0_skipped_virt_users + 1) &
                              ~R_MATRIX_ALIGN_MASK);
      R0_align = ((R0_skipped_virt_users + 1) & R_MATRIX_ALIGN_MASK) + 1;

      gen_R_matrices (
        R_sums1 + (R0_skipped_virt_users + 2),
        bf_scalep,
        inv_scalep + (R0_skipped_virt_users + 2),
        scalep + (R0_skipped_virt_users + 2),
        R0_lower_bf + R0_align,
```

```
        R0 upper bf + R0 align,
        R0_virt_users - 1
      );

      R0_upper_bf[ R0_align - 1 ] = 0;  /* zero diagonal element */

      R0 lower bf += R0 tcols;
      R0_upper_bf += R0_tcols;

      /*
       *  create ptov map[i] number of 32-element dot products involving
       *  X_bf[i] and corr_1_bf[i][j] where 0 < j < ptov_map[i]
       */
      gen R sums2 (
        X bf,
        corr 1 bf,
        corr 1 bf + (tot_virt_users * NUM_FINGERS_SQUARED),
        ptov map,
        R sums0,
        R sums1,
        tot_phys_users
      );

      /*
       *  scale the results and create two output rows (1 per matrix)
       */
      gen R matrices (
        R sums0,
        bf scalep,
        inv scalep + (R0_skipped_virt_users + 1),
        scalep,
        R1 trans_bf,
        R1m bf,
        tot_virt_users
      );

      R1 trans bf += R1 tcols;
      R1m_bf += R1_tcols;

      gen R matrices (
        R sums1,
        bf scalep,
        inv scalep + (R0_skipped_virt_users + 2),
        scalep,
        R1 trans_bf,
        R1m bf,
        tot_virt_users
      );

      R1 trans bf += R1 tcols;
      R1m_bf += R1_tcols;

      corr 0 bf += (((2 * R0 virt users) - 1) * NUM FINGERS SQUARED);
      corr 1 bf += ((2 * tot_virt_users) * NUM_FINGERS_SQUARED);
      R0 ptov map[i] -= 2;
      R0 virt users -= 2;
      R0_skipped_virt_users += 2;
    }

    if ( iv <= last_virt_user ) {

      bump = R0 ptov_map[ i ] ? 0 : 1;
      gen R sums (
        X bf + ((i + bump) * NUM_FINGERS_SQUARED),
        corr 0 bf,
        R0 ptov_map + i + bump,
        R_sums0 + (R0_skipped_virt_users + 1),
```

rmats.c 2/23/2001

```
        tot_phys_users - i - bump
      );

      R0 tcols = R1 tcols - (R0 skipped_virt users & ~R MATRIX_ALIGN_MASK);
      R0_align = (R0_skipped_virt_users & R_MATRIX_ALIGN_MASK) + 1;

      gen R matrices (
        R sums0 + (R0_skipped_virt_users + 1),
        bf scalep,
        inv scalep + (R0 skipped virt users + 1),
        scalep + (R0 skipped virt_users + 1),
        R0 lower bf + R0 align,
        R0 upper bf + R0_align,
        R0_virt_users
      );

      R0_upper_bf[ R0_align - 1 ] = 0;  /* zero diagonal element */

      R0 lower bf += R0 tcols;
      R0_upper_bf += R0_tcols;

      /*
       *  create ptov map[i] number of 32-element dot products involving
       *  X_bf[i] and corr_1_bf[i][j] where 0 < j < ptov_map[i]
       */
      gen R sums (
        X bf,
        corr 1 bf,
        ptov map,
        R sums0,
        tot_phys_users
      );

      /*
       *  scale the results and create two output rows (1 per matrix)
       */
      gen R matrices (
        R sums0,
        bf scalep,
        inv scalep + (R0_skipped_virt_users + 1),
        scalep,
        R1 trans_bf,
        R1m bf,
        tot_virt_users
      );

      R1 trans bf += R1 tcols;
      R1m_bf += R1_tcols;

      corr 0 bf += (R0 virt users * NUM FINGERS SQUARED);
      corr 1 bf += (tot_virt_users * NUM_FINGERS_SQUARED);
      R0 ptov map[i] -= 1;
      R0 virt users -= 1;
      R0_skipped_virt_users += 1;
    }
    start_virt_user = 0;                    /* for all subsequent passes */
  }

#if DO CALC STATS
  printf( "max R value = %f\n", max_R_value );
  if ( max_R_value > 127.0 )
    printf ( "***** OVERFLOW *****\n" );
#endif
}

#if COMPILE_C
```

rmats.c 2/23/2001

```
void  gen_X_row (
      COMPLEX_BF16  *mpath1_bf,
      COMPLEX_BF16  *mpath2_bf,
      COMPLEX_BF16  *X_bf,
      int  phys_index,
      int  tot_phys_users
    )

{
  COMPLEX_BF16  *in_mpath1p, *in_mpath2p;
  COMPLEX_BF16  *out_mpath1p, *out_mpath2p;
  int  i, j, q, q1;
  BF32  s1r, s1i, s2r, s2i;
  BF32  a1r, a1i, a2r, a2i;
  BF32  cr, ci;

  out_mpath1p = mpath1_bf + (phys_index * NUM_FINGERS);
  out_mpath2p = mpath2_bf + (phys_index * NUM_FINGERS);

  for ( i = 0; i < tot_phys_users; i++ ) {

    in_mpath1p = mpath1_bf + (i * NUM_FINGERS);  /* 4 complex values */
    in_mpath2p = mpath2_bf + (i * NUM_FINGERS);  /* 4 complex values */

    j = 0;
    for ( q1 = 0; q1 < NUM_FINGERS; q1++ ) {

      s1r = (BF32)out_mpath1p[q1].real;
      s1i = (BF32)out_mpath1p[q1].imag;
      s2r = (BF32)out_mpath2p[q1].real;
      s2i = (BF32)out_mpath2p[q1].imag;

      for ( q = 0; q < NUM_FINGERS; q++ ) {

        a1r = (BF32)in_mpath1p[q].real;
        a1i = (BF32)in_mpath1p[q].imag;
        a2r = (BF32)in_mpath2p[q].real;
        a2i = (BF32)in_mpath2p[q].imag;

        cr = (a1r * s1r) + (a1i * s1i);
        ci = (a1r * s1i) - (a1i * s1r);
        cr += (a2r * s2r) + (a2i * s2i);
        ci += (a2r * s2i) - (a2i * s2r);

        X_bf[i * NUM_FINGERS_SQUARED + j].real = (BF16)(cr >> 16);
        X_bf[i * NUM_FINGERS_SQUARED + j].imag = (BF16)(ci >> 16);
        ++j;
      }
    }
  }
}

void  gen_R_sums (
       COMPLEX_BF16  *X_bf,
       COMPLEX_BF8  *corr_bf,
       uchar  *ptov_map,
       BF32  *R_sums,
       int  num_phys_users
     )

{
  int  i, j, k;
  BF32  sum;

  for ( i = 0; i < num_phys_users; i++ ) {
    for ( j = 0; j < (int)ptov_map[i]; j++ ) {
      sum = 0;
```

rmats.c 2/23/2001

```
      for ( k = 0; k < 16; k++ ) {
        sum += (BF32)X bf[k].real * (BF32)corr bf->real;
        sum += (BF32)X_bf[k].imag * (BF32)corr_bf->imag;
        ++corr_bf;
      }
      *R_sums++ = sum;
    }
    X_bf += NUM_FINGERS_SQUARED;
  }
}

void  gen R sums2 (
        COMPLEX BF16  *X bf,
        COMPLEX BF8  *corra bf,
        COMPLEX BF8  *corrb_bf,
        uchar  *ptov map,
        BF32  *R sumsa,
        BF32  *R sumsb,
        int  num_phys_users
     )

{
  int  i, j, k;
  BF32  suma, sumb;

  for ( i = 0; i < num phys users; i++ ) {
    for ( j = 0; j < (int)ptov_map[i]; j++ ) {
      suma = 0;
      sumb = 0;
      for ( k = 0; k < 16; k++ ) {
        suma += (BF32)X bf[k].real * (BF32)corra bf->real;
        suma += (BF32)X bf[k].imag * (BF32)corra bf->imag;
        sumb += (BF32)X bf[k].real * (BF32)corrb bf->real;
        sumb += (BF32)X_bf[k].imag * (BF32)corrb_bf->imag;
        ++corra bf;
        ++corrb_bf;
      }
      *R sumsa++ = suma;
      *R_sumsb++ = sumb;
    }
    X_bf += NUM_FINGERS_SQUARED;
  }
}

void  gen R matrices (
        BF32  *R sums,
        float  *bf scalep,
        float  *inv scalep,
        float  *scalep,
        BF8  *no scale row bf,
        BF8  *scale row bf,
        int  num_virt_users
      )

{
  int  i;
  float  bf_scale, fsum, fsum_scale, inv_scale, scale;

  bf scale = *bf scalep;
  inv_scale = *inv_scalep;

  for ( i = 0; i < num_virt_users; i++ ) {
    scale = scalep[i];
    fsum = (float)(R sums[i]);
    fsum *= bf_scale;

    fsum_scale = fsum * inv_scale;
```

```
    fsum_scale *= scale;

#if DO_CALC_STATS
    UPDATE_MAX( fsum_scale, max_R_value )
    UPDATE_MAX( fsum, max_R_value )
#endif

#if DO_SQUELCH
    if ( FABS( fsum_scale ) <= SQUELCH_THRESH ) fsum_scale = 0.0;
    if ( FABS( fsum ) <= SQUELCH_THRESH ) fsum = 0.0;
#endif

#if DO_SATURATE
    SATURATE( fsum_scale )
    SATURATE( fsum )
#endif

    no_scale_row_bf[i] = BF8_FIX( fsum );
    scale_row_bf[i] = BF8_FIX( fsum_scale );
  }
}

#endif                          /* COMPILE_C */
```

dotpr3_8bit.mac 2/23/2001

```
/*---------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version   ---|
|---------------------------------------------------------------|
|                                                               |
|  File Name:    dotpr3_8bit.mac                                |
|  Description:  Source code for routine which computes three   |
|                dot products, combining the three sums prior   |
|                to exit.                                       |
|                                                               |
|             Mercury Computer Systems, Inc.                    |
|             Copyright (c) 2000  All rights reserved           |
|                                                               |
|   Revision       Date      Engineer  Reason                   |
|   --------       ----      --------  --------                 |
|     0.0         000510       fpl     Created                  |
|     0.1         000521       fpl     Added num cached_rows    |
|     0.2         000521       fpl     Changed to fixed point   |
|     0.3         000605       fpl     Changed to .k file       |
|     0.4         000926        jg     Back to .mac and no dsts |
+--------------------------------------------------------------*/

#include "salppc.inc"

#define LVX_BT( vT, rA, rB )          LVX( vT, rA, rB )

#define FUNC ENTRY                    dotpr3 8bit
#define VMSUM( vT, vA, vB, vC )       VMSUMMBM( vT, vA, vB, vC )
#define LOOP COUNT SHIFT 6
#define HALF BLOCK BIT 0x20
#define QUARTER_BLOCK_BIT 0x10

#define LOOP_BLOCK_SIZE 64

/**
 Input parameters
**/
#define bt1mptr r3
#define r1ptr   r4
#define r0ptr   r5
#define r1mptr  r6
#define C       r7
#define N       r8
#define hat_tc  r9
/**
 Local loop registers
**/
#define bt0ptr r10
#define bt1ptr r11
#define index1 r12
#define index2 r13

#define index3 r0
#define icount hat_tc

/**
 G4 registers
**/
#define rq10 v0
#define rq11 v1
#define rq12 v2
#define rq13 v3
#define zero v3

#define rq00 v4
#define rq01 v5
#define rq02 v6
```

dotpr3_8bit.mac 2/23/2001

```
#define rq03 v7

#define rq1m0 v8
#define rq1m1 v9
#define rq1m2 v10
#define rq1m3 v11

#define bt1m0 v12
#define bt1m1 v13
#define bt1m2 v14
#define bt1m3 v15

#define bt10 v16
#define bt11 v17
#define bt12 v18
#define bt13 v19

#define bt00 v20
#define bt01 v21
#define bt02 v22
#define bt03 v23

#define sum0 v24
#define sum1 v25
#define sum2 v26
#define sum3 v27

/**
 Begin code text
 Setup loop registers, test for zero N
**/
FUNC PROLOG
ENTRY 7( FUNC_ENTRY, bt1mptr, r1ptr, r0ptr, r1mptr, C, N, hat_tc )
  SAVE r13
  USE_THRU_v27( VRSAVE_COND )
/**
 Load up local loop registers
**/
   ADD(bt0ptr, bt1mptr, hat_tc)
   VXOR(sum0, sum0, sum0)
   ADD(bt1ptr, bt0ptr, hat_tc)

   LI(index1, 16)
   VXOR(sum1, sum1, sum1)
   LI(index2, 32)
   VXOR(sum2, sum2, sum2)
   LI(index3, 48)
   VXOR(sum3, sum3, sum3)
   SRWI C(icount, N, LOOP_COUNT_SHIFT) /* 32 sum updates per loop trip */
   BEQ(do_half_block)
/**
 Loop entry code
**/
   LVX( rq10, 0, r1ptr )
   LVX( rq11, r1ptr, index1 )
   LVX( rq12, r1ptr, index2 )
   LVX( rq13, r1ptr, index3 )
   DECR_C(icount)

   LVX BT( bt1m0, 0, bt1mptr )
   LVX BT( bt1m1, bt1mptr, index1 )
   ADDI(r1ptr, r1ptr, LOOP_BLOCK SIZE)
   LVX BT( bt1m2, bt1mptr, index2 )
   LVX BT( bt1m3, bt1mptr, index3 )
   ADDI(bt1mptr, bt1mptr, LOOP_BLOCK_SIZE)
   BR( mid_loop )
/**
```

dotpr3_8bit.mac 2/23/2001

```
 Loop computes three dot products held in 16 parts
**/
LABEL( loop )
/* { */
    LVX( rq10, 0, r1ptr )
    VMSUM( sum0, rq1m0, bt10, sum0 )
    LVX( rq11, r1ptr, index1 )
    VMSUM( sum1, rq1m1, bt11, sum1 )
    LVX( rq12, r1ptr, index2 )
    VMSUM( sum2, rq1m2, bt12, sum2 )
    LVX( rq13, r1ptr, index3 )
    DECR_C(icount)

    LVX BT( bt1m0, 0, bt1mptr )
    VMSUM( sum3, rq1m3, bt13, sum3 )
    LVX BT( bt1m1, bt1mptr, index1 )
    ADDI(r1ptr, r1ptr, LOOP_BLOCK SIZE)
    LVX BT( bt1m2, bt1mptr, index2 )
    LVX BT( bt1m3, bt1mptr, index3 )
    ADDI(bt1mptr, bt1mptr, LOOP_BLOCK_SIZE)

LABEL( mid_loop )

    LVX( rq00, 0, r0ptr )
    VMSUM( sum0, rq10, bt1m0, sum0 )
    LVX( rq01, r0ptr, index1 )
    VMSUM( sum1, rq11, bt1m1, sum1 )
    LVX( rq02, r0ptr, index2 )
    VMSUM( sum2, rq12, bt1m2, sum2 )
    LVX( rq03, r0ptr, index3 )

    LVX BT( bt00, 0, bt0ptr )
    VMSUM( sum3, rq13, bt1m3, sum3 )
    LVX BT( bt01, bt0ptr, index1 )
    ADDI(r0ptr, r0ptr, LOOP BLOCK SIZE)
    LVX BT( bt02, bt0ptr, index2 )
    LVX BT( bt03, bt0ptr, index3 )
    ADDI(bt0ptr, bt0ptr, LOOP_BLOCK_SIZE)

    LVX( rq1m0, 0, r1mptr )
    VMSUM( sum0, rq00, bt00, sum0 )
    LVX( rq1m1, r1mptr, index1 )
    VMSUM( sum1, rq01, bt01, sum1 )
    LVX( rq1m2, r1mptr, index2 )
    VMSUM( sum2, rq02, bt02, sum2 )
    LVX( rq1m3, r1mptr, index3 )

    LVX BT( bt10, 0, bt1ptr )
    VMSUM( sum3, rq03, bt03, sum3 )
    LVX BT( bt11, bt1ptr, index1 )
    ADDI(r1mptr, r1mptr, LOOP BLOCK_SIZE)
    LVX BT( bt12, bt1ptr, index2 )
    LVX BT( bt13, bt1ptr, index3 )
    ADDI(bt1ptr, bt1ptr, LOOP_BLOCK_SIZE)
/* } */
  BNE( loop )
/**
 Loop exit code
**/
   VMSUM( sum0, rq1m0, bt10, sum0 )
   VMSUM( sum1, rq1m1, bt11, sum1 )
   VMSUM( sum2, rq1m2, bt12, sum2 )
   VMSUM( sum3, rq1m3, bt13, sum3 )
/**
 Remainders
**/
LABEL(do_half_block)
```

```
   ANDI C( icount, N, HALF_BLOCK_BIT )
   BEQ(do quarter block)
   LVX( rq10, 0, r1ptr )
   LVX( rq11, r1ptr, index1 )
   LVX BT( bt1m0, 0, bt1mptr )
   LVX BT( bt1m1, bt1mptr, index1 )
   ADDI(r1ptr, r1ptr, (LOOP BLOCK SIZE >> 1) )
   ADDI(bt1mptr, bt1mptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum0, rq10, bt1m0, sum0 )
   VMSUM( sum1, rq11, bt1m1, sum1 )

   LVX( rq00, 0, r0ptr )
   LVX( rq01, r0ptr, index1 )
   LVX BT( bt00, 0, bt0ptr )
   LVX BT( bt01, bt0ptr, index1 )
   ADDI(r0ptr, r0ptr, (LOOP BLOCK SIZE >> 1) )
   ADDI(bt0ptr, bt0ptr, (LOOP_BLOCK_SIZE >> 1) ).

   VMSUM( sum0, rq00, bt00, sum0 )
   VMSUM( sum1, rq01, bt01, sum1 )

   LVX( rq1m0, 0, r1mptr )
   LVX( rq1m1, r1mptr, index1 )
   LVX BT( bt10, 0, bt1ptr )
   LVX BT( bt11, bt1ptr, index1 )
   ADDI(r1mptr, r1mptr, (LOOP BLOCK SIZE >> 1) )
   ADDI(bt1ptr, bt1ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum0, rq1m0, bt10, sum0 )
   VMSUM( sum1, rq1m1, bt11, sum1 )

LABEL(do quarter block)
   ANDI C( icount, N, QUARTER_BLOCK_BIT )
   BEQ(combine)
   LVX( rq10, 0, r1ptr )
   LVX BT( bt1m0, 0, bt1mptr )
   VMSUM( sum0, rq10, bt1m0, sum0 )

   LVX( rq00, 0, r0ptr )
   LVX BT( bt00, 0, bt0ptr )
   VMSUM( sum0, rq00, bt00, sum0 )

   LVX( rq1m0, 0, r1mptr )
   LVX BT( bt10, 0, bt1ptr )
   VMSUM( sum0, rq1m0, bt10, sum0 )
/**
 Combine sums and return
**/
LABEL(combine)
   VXOR( zero, zero, zero )
   VADDSWS( sum0, sum0, sum1 ) /* s00 s01 s02 s03 */
   VADDSWS( sum2, sum2, sum3 ) /* s22 s21 s22 s23 */
   VADDSWS( sum0, sum0, sum2 ) /* s00 s01 s02 s03 */
   VSUMSWS( sum0, sum0, zero ) /* xxx xxx xxx s00 */
   VSPLTW( sum0, sum0, 3 )     /* s00 s00 s00 s00 */
   STVEWX( sum0, 0, C )
/**
 Return
**/
LABEL( ret )
  FREE THRU_v27( VRSAVE_COND )
  REST r13
  RETURN
FUNC_EPILOG
```

dotpr6_8bit.mac 3/9/2001

```
/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version   ---|
|---------------------------------------------------------------|
|                                                               |
|  File Name:    dotpr6_8bit.mac                                |
|  Description:  Source code for routine which computes six     |
|                dot products, combining the six sums prior     |
|                into two outputs prior to exit.                |
|                                                               |
|             Mercury Computer Systems, Inc.                    |
|             Copyright (c) 2000  All rights reserved           |
|                                                               |
|   Revision       Date      Engineer  Reason                   |
|   --------       ----      --------  --------                 |
|     0.0         000510       fpl    Created                   |
|     0.1         000521       fpl    Changed to fixed point    |
|     0.2         000521       fpl    Added num cached rows     |
|     0.3         000605       fpl    Changed to .k file        |
|     0.4         000926        jg    Back to .mac and no dsts  |
+--------------------------------------------------------------*/

#include "salppc.inc"

#define LVX_BT( vT, rA, rB )           LVX( vT, rA, rB )

#define FUNC ENTRY                     dotpr6 8bit
#define VMSUM( vT, vA, vB, vC )        VMSUMMBM( vT, vA, vB, vC )
#define LOOP COUNT SHIFT 6
#define HALF BLOCK BIT 0x20
#define QUARTER_BLOCK_BIT 0x10

#define LOOP_BLOCK_SIZE 64

/**
 Input parameters
**/
#define bt1mptr r3
#define r1ptr   r4
#define r0ptr   r5
#define r1mptr  r6
#define C       r7
#define N       r8
#define hat_tc  r9
/**
 Local loop registers
**/
#define bt0ptr r10
#define bt1ptr r11
#define bt2ptr r12
#define index1 r13
#define index2 r14

#define index3 r0
#define icount hat_tc

/**
 G4 registers
**/
#define rq10 v0
#define rq11 v1
#define rq12 v2
#define rq13 v3
#define zero v3

#define rq00 v4
#define rq01 v5
```

```
dotpr6_8bit.mac                                                    3/9/2001

#define rq02 v6
#define rq03 v7

#define rqlm0 v8
#define rqlm1 v9
#define rqlm2 v10
#define rqlm3 v11

#define btlm0 v12
#define btlm1 v13
#define btlm2 v14
#define btlm3 v15

#define bt10 v12
#define bt11 v13
#define bt12 v14
#define bt13 v15

#define bt00 v16
#define bt01 v17
#define bt02 v18
#define bt03 v19

#define bt20 v16
#define bt21 v17
#define bt22 v18
#define bt23 v19

#define sum00 v20
#define sum01 v21
#define sum02 v22
#define sum03 v23

#define sum10 v24
#define sum11 v25
#define sum12 v26
#define sum13 v27
/**
 Begin code text
**/
FUNC PROLOG
ENTRY 7( FUNC ENTRY, btlmptr, r1ptr, r0ptr, r1mptr, C, N, hat_tc )
  SAVE r13 r14
  USE_THRU_v27( VRSAVE_COND )
/**
 Load up local loop registers
**/
   ADD(bt0ptr, btlmptr, hat tc)
   VXOR(sum00, sum00, sum00)
   ADD(bt1ptr, bt0ptr, hat_tc)
   LI(index1, 16)
   ADD(bt2ptr, bt1ptr, hat_tc)

   VXOR(sum01, sum01, sum01)
   LI(index2, 32)
   VXOR(sum02, sum02, sum02)
   LI(index3, 48)
   VXOR(sum03, sum03, sum03)

   VXOR(sum10, sum10, sum10)
   VXOR(sum11, sum11, sum11)
   VXOR(sum12, sum12, sum12)
   VXOR(sum13, sum13, sum13)
   SRWI C(icount, N, LOOP_COUNT_SHIFT)
   BEQ(do_half_block)
/**
 Loop entry code
```

dotpr6_8bit.mac 3/9/2001

```
**/
   LVX_BT( bt1m0, 0, bt1mptr )
   DECR_C(icount)
   LVX_BT( bt1m1, bt1mptr, index1 )
   LVX_BT( bt1m2, bt1mptr, index2 )
   LVX_BT( bt1m3, bt1mptr, index3 )

   LVX( rq10, 0, r1ptr )
   LVX( rq11, r1ptr, index1 )
   ADDI(bt1mptr, bt1mptr,  LOOP_BLOCK_SIZE)
   LVX( rq12, r1ptr, index2 )
   LVX( rq13, r1ptr, index3 )
   BR( mid_loop )
/**
 Loop computes three dot products held in 16 parts
**/
LABEL( loop )
/* { */
    LVX_BT( bt1m0, 0, bt1mptr )
    VMSUM( sum10, rq1m0, bt20, sum10 )
    LVX_BT( bt1m1, bt1mptr, index1 )
    VMSUM( sum11, rq1m1, bt21, sum11 )
    LVX_BT( bt1m2, bt1mptr, index2 )
    DECR_C(icount)
    VMSUM( sum12, rq1m2, bt22, sum12 )
    LVX_BT( bt1m3, bt1mptr, index3 )

    LVX( rq10, 0, r1ptr )
    VMSUM( sum13, rq1m3, bt23, sum13 )
    LVX( rq11, r1ptr, index1 )
    LVX( rq12, r1ptr, index2 )
    ADDI(bt2ptr, bt2ptr,  LOOP_BLOCK_SIZE)
    LVX( rq13, r1ptr, index3 )
    ADDI(bt1mptr, bt1mptr,  LOOP_BLOCK_SIZE)

LABEL( mid_loop )

    LVX_BT( bt00, 0, bt0ptr )
    VMSUM( sum00, rq10, bt1m0, sum00 )
    LVX_BT( bt01, bt0ptr, index1 )
    VMSUM( sum01, rq11, bt1m1, sum01 )
    LVX_BT( bt02, bt0ptr, index2 )
    VMSUM( sum02, rq12, bt1m2, sum02 )
    LVX_BT( bt03, bt0ptr, index3 )
    ADDI(r1ptr, r1ptr,  LOOP_BLOCK_SIZE)

    LVX( rq00, 0, r0ptr )
    VMSUM( sum03, rq13, bt1m3, sum03 )
    LVX( rq01, r0ptr, index1 )
    VMSUM( sum10, rq10, bt00, sum10 )
    LVX( rq02, r0ptr, index2 )
    VMSUM( sum11, rq11, bt01, sum11 )
    ADDI(bt0ptr, bt0ptr,  LOOP_BLOCK_SIZE)
    VMSUM( sum12, rq12, bt02, sum12 )
    LVX( rq03, r0ptr, index3 )
    VMSUM( sum13, rq13, bt03, sum13 )

    LVX_BT( bt10, 0, bt1ptr )
    VMSUM( sum00, rq00, bt00, sum00 )
    LVX_BT( bt11, bt1ptr, index1 )
    ADDI(r0ptr, r0ptr,  LOOP_BLOCK_SIZE)
    LVX_BT( bt12, bt1ptr, index2 )
    VMSUM( sum01, rq01, bt01, sum01 )
    LVX_BT( bt13, bt1ptr, index3 )
    VMSUM( sum02, rq02, bt02, sum02 )

    LVX( rq1m0, 0, r1mptr )
```

```
    VMSUM( sum03, rq03, bt03, sum03 )
    ADDI(bt1ptr, bt1ptr,  LOOP BLOCK SIZE)
    VMSUM( sum10, rq00, bt10, sum10 )
    LVX( rq1m1, r1mptr, index1 )
    VMSUM( sum11, rq01, bt11, sum11 )
    LVX( rq1m2, r1mptr, index2 )
    VMSUM( sum12, rq02, bt12, sum12 )
    LVX( rq1m3, r1mptr, index3 )
    ADDI(r1mptr, r1mptr,  LOOP_BLOCK_SIZE)

    LVX BT( bt20, 0, bt2ptr )
    VMSUM( sum13, rq03, bt13, sum13 )
    LVX BT( bt21, bt2ptr, index1 )
    VMSUM( sum00, rq1m0, bt10, sum00 )
    LVX BT( bt22, bt2ptr, index2 )
    VMSUM( sum01, rq1m1, bt11, sum01 )
    LVX BT( bt23, bt2ptr, index3 )
    VMSUM( sum02, rq1m2, bt12, sum02 )
    VMSUM( sum03, rq1m3, bt13, sum03 )
/* } */
  BNE( loop )
/**
 Loop exit code
**/
   VMSUM( sum10, rq1m0, bt20, sum10 )
   VMSUM( sum11, rq1m1, bt21, sum11 )
   ADDI(bt2ptr, bt2ptr,  LOOP_BLOCK_SIZE)
   VMSUM( sum12, rq1m2, bt22, sum12 )
   VMSUM( sum13, rq1m3, bt23, sum13 )
/**
 Remainders
**/
LABEL(do half block)
   ANDI C( icount, N, HALF_BLOCK_BIT )
   BEQ(do_quarter_block)

   LVX BT( bt1m0, 0, bt1mptr )
   LVX BT( bt1m1, bt1mptr, index1 )
   ADDI(bt1mptr, bt1mptr, (LOOP_BLOCK_SIZE >> 1) )

   LVX( rq10, 0, r1ptr )
   LVX( rq11, r1ptr, index1 )
   ADDI(r1ptr, r1ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum00, rq10, bt1m0, sum00 )
   VMSUM( sum01, rq11, bt1m1, sum01 )

   LVX BT( bt00, 0, bt0ptr )
   LVX BT( bt01, bt0ptr, index1 )
   ADDI(bt0ptr, bt0ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum10, rq10, bt00, sum10 )
   VMSUM( sum11, rq11, bt01, sum11 )

   LVX( rq00, 0, r0ptr )
   LVX( rq01, r0ptr, index1 )
   ADDI(r0ptr, r0ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum00, rq00, bt00, sum00 )
   VMSUM( sum01, rq01, bt01, sum01 )

   LVX BT( bt10, 0, bt1ptr )
   LVX BT( bt11, bt1ptr, index1 )
   ADDI(bt1ptr, bt1ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum10, rq00, bt10, sum10 )
   VMSUM( sum11, rq01, bt11, sum11 )
```

```
    LVX( rqlm0, 0, rlmptr )
    LVX( rqlm1, rlmptr, index1 )
    ADDI(rlmptr, rlmptr, (LOOP_BLOCK_SIZE >> 1) )

    VMSUM( sum00, rqlm0, bt10, sum00 )
    VMSUM( sum01, rqlm1, bt11, sum01 )

    LVX BT( bt20, 0, bt2ptr )
    LVX BT( bt21, bt2ptr, index1 )
    ADDI(bt2ptr, bt2ptr, (LOOP_BLOCK_SIZE >> 1) )

    VMSUM( sum10, rqlm0, bt20, sum10 )
    VMSUM( sum11, rqlm1, bt21, sum11 )

LABEL(do quarter block)
    ANDI C( icount, N, QUARTER_BLOCK_BIT )
    BEQ(combine)

    LVX BT( btlm0, 0, btlmptr )
    LVX( rq10, 0, rlptr )
    VMSUM( sum00, rq10, btlm0, sum00 )

    LVX BT( bt00, 0, bt0ptr )
    VMSUM( sum10, rq10, bt00, sum10 )
    LVX( rq00, 0, r0ptr )
    VMSUM( sum00, rq00, bt00, sum00 )

    LVX BT( bt10, 0, bt1ptr )
    VMSUM( sum10, rq00, bt10, sum10 )
    LVX( rqlm0, 0, rlmptr )
    VMSUM( sum00, rqlm0, bt10, sum00 )
    LVX BT( bt20, 0, bt2ptr )
    VMSUM( sum10, rqlm0, bt20, sum10 )
/**
 Combine sums and return
**/
LABEL(combine)
    VXOR( zero, zero, zero )
    VADDSWS( sum00, sum00, sum01 )   /* s00 s01 s02 s03 */
    VADDSWS( sum10, sum10, sum11 )
    VADDSWS( sum02, sum02, sum03 )   /* s22 s21 s22 s23 */
    VADDSWS( sum12, sum12, sum13 )
    VADDSWS( sum00, sum00, sum02 )   /* s00 s01 s02 s03 */
    VADDSWS( sum10, sum10, sum12 )

    VSUMSWS( sum00, sum00, zero )    /* xxx xxx xxx s00 */
    VSUMSWS( sum10, sum10, zero )
    VSPLTW( sum00, sum00, 3 )        /* s00 s00 s00 s00 */
    STVEWX( sum00, 0, C )
    ADDI( C, C, 4 )
    VSPLTW( sum10, sum10, 3 )
    STVEWX( sum10, 0, C )
/**
 Return
**/
LABEL( ret )
  FREE THRU v27( VRSAVE_COND )
  REST r13_r14
  RETURN
FUNC_EPILOG
```

dotpr9_8bit.mac 2/23/2001

```
/*------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version  ---|
|--------------------------------------------------------------|
|                                                              |
|  File Name:    dotpr9_8bit.mac                               |
|  Description:  Source code for routine which computes nine   |
|                dot products, combining the nine sums prior   |
|                into three outputs prior to exit.             |
|                                                              |
|             Mercury Computer Systems, Inc.                   |
|             Copyright (c) 2000  All rights reserved          |
|                                                              |
|   Revision       Date      Engineer  Reason                  |
|   --------       ----      --------  --------                |
|     0.0         000510       fpl     Created                 |
|     0.1         000512       fpl     Added num cached_rows   |
|     0.2         000521       fpl     Changed to fixed point  |
|     0.3         000605       fpl     Changed to .k file      |
|     0.4         000926        jg     Back to .mac and no dsts|
+------------------------------------------------------------*/

#include "salppc.inc"

#define LVX_BT( vT, rA, rB )           LVX( vT, rA, rB )

#define FUNC ENTRY                     dotpr9 8bit
#define VMSUM( vT, vA, vB, vC )        VMSUMMBM( vT, vA, vB, vC )
#define LOOP COUNT SHIFT 6
#define HALF BLOCK BIT 0x20
#define QUARTER_BLOCK_BIT 0x10

#define LOOP_BLOCK_SIZE 64

/**
 Input parameters
**/
#define bt1mptr r3
#define r1ptr   r4
#define r0ptr   r5
#define r1mptr  r6
#define C       r7
#define N       r8
#define hat_tc  r9
/**
 Local loop registers
**/
#define bt0ptr r10
#define bt1ptr r11
#define bt2ptr r12
#define bt3ptr r13
#define index1 r14
#define index2 r15

#define index3 r0
#define icount hat_tc

/**
 G4 registers
**/
#define rq10 v0
#define rq11 v1
#define rq12 v2
#define rq13 v3
#define zero v3

#define bt30 v0
```

dotpr9_8bit.mac 2/23/2001

```
#define bt31 v1
#define bt32 v2
#define bt33 v3

#define rq00 v4
#define rq01 v5
#define rq02 v6
#define rq03 v7

#define rq1m0 v8
#define rq1m1 v9
#define rq1m2 v10
#define rq1m3 v11

#define bt1m0 v12
#define bt1m1 v13
#define bt1m2 v14
#define bt1m3 v15

#define bt10 v12
#define bt11 v13
#define bt12 v14
#define bt13 v15

#define bt00 v16
#define bt01 v17
#define bt02 v18
#define bt03 v19

#define bt20 v16
#define bt21 v17
#define bt22 v18
#define bt23 v19

#define sum00 v20
#define sum01 v21
#define sum02 v22
#define sum03 v23

#define sum10 v24
#define sum11 v25
#define sum12 v26
#define sum13 v27

#define sum20 v28
#define sum21 v29
#define sum22 v30
#define sum23 v31

/**
 Begin code text
**/
FUNC PROLOG
ENTRY 7( FUNC ENTRY, bt1mptr, r1ptr, r0ptr, r1mptr, C, N, hat_tc )
  SAVE r13 r15
  USE_THRU_v31( VRSAVE_COND )
/**
 Load up local loop registers
**/
   ADD(bt0ptr, bt1mptr, hat tc)
   VXOR(sum00, sum00, sum00)
   ADD(bt1ptr, bt0ptr, hat_tc)
   LI(index1, 16)
   ADD(bt2ptr, bt1ptr, hat tc)
   VXOR(sum01, sum01, sum01)
   ADD(bt3ptr, bt2ptr, hat_tc)
```

```
    LI(index2, 32)
    VXOR(sum02, sum02, sum02)
    LI(index3, 48)
    VXOR(sum03, sum03, sum03)

    VXOR(sum10, sum10, sum10)
    VXOR(sum11, sum11, sum11)
    VXOR(sum12, sum12, sum12)
    VXOR(sum13, sum13, sum13)

    VXOR(sum20, sum20, sum20)
    VXOR(sum21, sum21, sum21)
    VXOR(sum22, sum22, sum22)
    VXOR(sum23, sum23, sum23)
    SRWI C(icount, N, LOOP_COUNT_SHIFT)
    BEQ(do_half_block)
/**
 Loop entry code
**/
    LVX BT( bt1m0, 0, bt1mptr )
    LVX BT( bt1m1, bt1mptr, index1 )
    DECR C(icount)
    LVX BT( bt1m2, bt1mptr, index2 )
    LVX_BT( bt1m3, bt1mptr, index3 )

    LVX( rq10, 0, r1ptr )
    ADDI(bt1mptr, bt1mptr, LOOP_BLOCK_SIZE)
    LVX( rq11, r1ptr, index1 )
    LVX( rq12, r1ptr, index2 )
    LVX( rq13, r1ptr, index3 )
    LVX_BT( bt00, 0, bt0ptr )
    BR( mid_loop )
/**
 Nine dot products producing 3 sums:
  sum0 = (R1 * Bt1m) (R0 * Bt0) (R1m * Bt1)
  sum1 = (R1 * Bt0) (R0 * Bt1) (R1m * Bt2)
  sum2 = (R1 * Bt1) (R0 * Bt2) (R1m * Bt3)
**/
LABEL( loop )
/* { */
     LVX BT( bt1m0, 0, bt1mptr )
     VMSUM( sum20, rq1m0, bt30, sum20 ) /* R1m * Bt3 */
     LVX BT( bt1m1, bt1mptr, index1 )
     VMSUM( sum21, rq1m1, bt31, sum21 )
     LVX BT( bt1m2, bt1mptr, index2 )
     VMSUM( sum22, rq1m2, bt32, sum22 )
     LVX_BT( bt1m3, bt1mptr, index3 )

     LVX( rq10, 0, r1ptr )
     VMSUM( sum23, rq1m3, bt33, sum23 )
     ADDI(bt1mptr, bt1mptr, LOOP_BLOCK_SIZE)
     LVX( rq11, r1ptr, index1 )
     VMSUM( sum20, rq00, bt20, sum20 ) /* R0 * Bt2 */
     LVX( rq12, r1ptr, index2 )
     VMSUM( sum21, rq01, bt21, sum21 )
     DECR C(icount)
     VMSUM( sum22, rq02, bt22, sum22 )
     LVX( rq13, r1ptr, index3 )

     VMSUM( sum23, rq03, bt23, sum23 )
     LVX_BT( bt00, 0, bt0ptr )
/**
 Loop entry
**/
LABEL( mid_loop )
     VMSUM( sum00, rq10, bt1m0, sum00 ) /* R1 * Bt1m */
     LVX_BT( bt01, bt0ptr, index1 )
```

dotpr9_8bit.mac 2/23/2001

```
     ADDI(r1ptr, r1ptr, LOOP BLOCK SIZE)
     LVX BT( bt02, bt0ptr, index2 )
     VMSUM( sum01, rq11, bt1m1, sum01 )
     LVX_BT( bt03, bt0ptr, index3 )

     VMSUM( sum02, rq12, bt1m2, sum02 )
     LVX( rq00, 0, r0ptr )
     VMSUM( sum03, rq13, bt1m3, sum03 )
     ADDI(bt0ptr, bt0ptr, LOOP_BLOCK_SIZE)
     VMSUM( sum10, rq10, bt00, sum10 ) /* R1 * Bt0 */
     LVX( rq01, r0ptr, index1 )
     VMSUM( sum11, rq11, bt01, sum11 )
     LVX( rq02, r0ptr, index2 )
     VMSUM( sum12, rq12, bt02, sum12 )
     LVX( rq03, r0ptr, index3 )

     ADDI(r0ptr, r0ptr, LOOP BLOCK SIZE)
     VMSUM( sum13, rq13, bt03, sum13 )
     LVX BT( bt10, 0, bt1ptr )
     LVX BT( bt11, bt1ptr, index1 )
     VMSUM( sum00, rq00, bt00, sum00 ) /* R0 * Bt0 */
     LVX BT( bt12, bt1ptr, index2 )
     VMSUM( sum01, rq01, bt01, sum01 )
     LVX_BT( bt13, bt1ptr, index3 )

     VMSUM( sum02, rq02, bt02, sum02 )
     VMSUM( sum03, rq03, bt03, sum03 )
     LVX( rq1m0, 0, r1mptr )
     VMSUM( sum20, rq10, bt10, sum20 ) /* R1 * Bt1 */
     LVX( rq1m1, r1mptr, index1 )
     VMSUM( sum21, rq11, bt11, sum21 )
     LVX( rq1m2, r1mptr, index2 )
     ADDI(bt1ptr, bt1ptr, LOOP BLOCK_SIZE)
     LVX( rq1m3, r1mptr, index3 )

     VMSUM( sum22, rq12, bt12, sum22 )
     LVX BT( bt20, 0, bt2ptr )
     VMSUM( sum23, rq13, bt13, sum23 )
     LVX BT( bt21, bt2ptr, index1 )
     VMSUM( sum10, rq00, bt10, sum10 ) /* R0 * Bt1 */
     ADDI(r1mptr, r1mptr, LOOP_BLOCK_SIZE)
     VMSUM( sum11, rq01, bt11, sum11 )
     LVX BT( bt22, bt2ptr, index2 )
     VMSUM( sum12, rq02, bt12, sum12 )
     LVX_BT( bt23, bt2ptr, index3 )

     VMSUM( sum13, rq03, bt13, sum13 )
     LVX BT( bt30, 0, bt3ptr )
     LVX BT( bt31, bt3ptr, index1 )
     VMSUM( sum00, rq1m0, bt10, sum00 ) /* R1m * Bt1 */
     LVX BT( bt32, bt3ptr, index2 )
     VMSUM( sum01, rq1m1, bt11, sum01 )
     LVX_BT( bt33, bt3ptr, index3 )

     VMSUM( sum02, rq1m2, bt12, sum02 )
     VMSUM( sum03, rq1m3, bt13, sum03 )
     ADDI(bt2ptr, bt2ptr, LOOP BLOCK SIZE)
     VMSUM( sum10, rq1m0, bt20, sum10 ) /* R1m * Bt2 */
     VMSUM( sum11, rq1m1, bt21, sum11 )
     ADDI(bt3ptr, bt3ptr, LOOP BLOCK SIZE)
     VMSUM( sum12, rq1m2, bt22, sum12 )
     VMSUM( sum13, rq1m3, bt23, sum13 )
/* } */
  BNE( loop )
/**
 Loop exit code
**/
```

```
   VMSUM( sum20, rq1m0, bt30, sum20 ) /* R1m * Bt3 */
   VMSUM( sum21, rq1m1, bt31, sum21 )
   VMSUM( sum22, rq1m2, bt32, sum22 )
   VMSUM( sum23, rq1m3, bt33, sum23 )
   VMSUM( sum20, rq00, bt20, sum20 ) /* R0 * Bt2 */
   VMSUM( sum21, rq01, bt21, sum21 )
   VMSUM( sum22, rq02, bt22, sum22 )
   VMSUM( sum23, rq03, bt23, sum23 )
/**
 Remainders
**/
LABEL(do half block)
   ANDI C( icount, N, HALF_BLOCK_BIT )
   BEQ(do_quarter_block)

   LVX BT( bt1m0, 0, bt1mptr )
   LVX BT( bt1m1, bt1mptr, index1 )
   ADDI(bt1mptr, bt1mptr, (LOOP_BLOCK_SIZE >> 1) )

   LVX( rq10, 0, r1ptr )
   LVX( rq11, r1ptr, index1 )
   ADDI(r1ptr, r1ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum00, rq10, bt1m0, sum00 ) /* R1 * Bt1m */
   VMSUM( sum01, rq11, bt1m1, sum01 )

   LVX BT( bt00, 0, bt0ptr )
   LVX BT( bt01, bt0ptr, index1 )
   ADDI(bt0ptr, bt0ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum10, rq10, bt00, sum10 ) /* R1 * Bt0 */
   VMSUM( sum11, rq11, bt01, sum11 )

   LVX( rq00, 0, r0ptr )
   LVX( rq01, r0ptr, index1 )
   ADDI(r0ptr, r0ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum00, rq00, bt00, sum00 ) /* R0 * Bt0 */
   VMSUM( sum01, rq01, bt01, sum01 )

   LVX BT( bt10, 0, bt1ptr )
   LVX BT( bt11, bt1ptr, index1 )
   ADDI(bt1ptr, bt1ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum20, rq10, bt10, sum20 ) /* R1 * Bt1 */
   VMSUM( sum21, rq11, bt11, sum21 )

   VMSUM( sum10, rq00, bt10, sum10 ) /* R0 * Bt1 */
   VMSUM( sum11, rq01, bt11, sum11 )

   LVX( rq1m0, 0, r1mptr )
   LVX( rq1m1, r1mptr, index1 )
   ADDI(r1mptr, r1mptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum00, rq1m0, bt10, sum00 ) /* R1m * Bt1 */
   VMSUM( sum01, rq1m1, bt11, sum01 )

   LVX BT( bt20, 0, bt2ptr )
   LVX BT( bt21, bt2ptr, index1 )
   ADDI(bt2ptr, bt2ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum20, rq00, bt20, sum20 ) /* R0 * Bt2 */
   VMSUM( sum21, rq01, bt21, sum21 )

   VMSUM( sum10, rq1m0, bt20, sum10 ) /* R1m * Bt2 */
   VMSUM( sum11, rq1m1, bt21, sum11 )
```

```
   LVX BT( bt30, 0, bt3ptr )
   LVX BT( bt31, bt3ptr, index1 )
   ADDI(bt3ptr, bt3ptr, (LOOP_BLOCK_SIZE >> 1) )

   VMSUM( sum20, rq1m0, bt30, sum20 ) /* R1m * Bt3 */
   VMSUM( sum21, rq1m1, bt31, sum21 )
/**
 four more sums
**/
LABEL(do quarter block)
   ANDI C( icount, N, QUARTER_BLOCK_BIT )
   BEQ(combine)

   LVX BT( bt1m0, 0, bt1mptr )
   LVX( rq10, 0, r1ptr )
   VMSUM( sum00, rq10, bt1m0, sum00 ) /* R1 * Bt1m */
   ADDI(bt1mptr, bt1mptr, 16)

   LVX BT( bt00, 0, bt0ptr )
   VMSUM( sum10, rq10, bt00, sum10 ) /* R1 * Bt0 */

   LVX( rq00, 0, r0ptr )
   VMSUM( sum00, rq00, bt00, sum00 ) /* R0 * Bt0 */

   LVX_BT( bt10, 0, bt1ptr )

   VMSUM( sum20, rq10, bt10, sum20 ) /* R1 * Bt1 */
   VMSUM( sum10, rq00, bt10, sum10 ) /* R0 * Bt1 */

   LVX( rq1m0, 0, r1mptr )
   VMSUM( sum00, rq1m0, bt10, sum00 ) /* R1m * Bt1 */

   LVX BT( bt20, 0, bt2ptr )
   VMSUM( sum20, rq00, bt20, sum20 ) /* R0 * Bt2 */
   VMSUM( sum10, rq1m0, bt20, sum10 ) /* R1m * Bt2 */

   LVX BT( bt30, 0, bt3ptr )
   VMSUM( sum20, rq1m0, bt30, sum20 ) /* R1m * Bt3 */
/**
 Combine sums and return
**/
LABEL(combine)
   VXOR( zero, zero, zero )
   VADDSWS( sum00, sum00, sum01 )
   VADDSWS( sum10, sum10, sum11 )
   VADDSWS( sum20, sum20, sum21 )

   VADDSWS( sum02, sum02, sum03 )
   VADDSWS( sum12, sum12, sum13 )
   VADDSWS( sum22, sum22, sum23 )

   VADDSWS( sum00, sum00, sum02 )
   VADDSWS( sum10, sum10, sum12 )
   VADDSWS( sum20, sum20, sum22 )

   VSUMSWS( sum00, sum00, zero )    /* xxx xxx xxx s00 */
   VSUMSWS( sum10, sum10, zero )
   VSUMSWS( sum20, sum20, zero )

   VSPLTW( sum00, sum00, 3 )        /* s00 s00 s00 s00 */
   STVEWX( sum00, 0, C )
   ADDI( C, C, 4 )
   VSPLTW( sum10, sum10, 3 )
   STVEWX( sum10, 0, C )
   ADDI( C, C, 4 )
   VSPLTW( sum20, sum20, 3 )
   STVEWX( sum20, 0, C )
```

dotpr9_8bit.mac 2/23/2001

```
/**
 Return
**/
LABEL( ret )
  FREE THRU v31( VRSAVE_COND )
  REST r13_r15
  RETURN
FUNC_EPILOG
```

fixed_cdotpr.mac 2/23/2001

```
#ifndef MCOS 55
#define MCOS_55 0
#endif
/*-------------------------------------------------------------+
/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- 603e Macro language Version  ---|
|---------------------------------------------------------------|
|                                                               |
|   File Name:    CDOTPR.MAC                                    |
|   Description:  Vector Single Precision Complex Dot Product   |
|   Entry/params: CDOTPR (A, I, B, J, C, N)                     |
|   Formula: C[0] = sum (A[mI]*B[mJ] - A[mI+1]*B[mJ+1])         |
|            C[1] = sum (A[mI]*B[mJ+1] + A[mI+1]*B[mJ])         |
|                   for m=0 to N-1                              |
|                                                               |
|              Mercury Computer Systems, Inc.                   |
|              Copyright (c) 1995  All rights reserved          |
|                                                               |
|    Revision      Date        Engineer Reason                  |
|    --------      ----        ----------------                 |
|      0.0        960502      fpl  Created                      |
|      0.1        960618      fpl  Added Esal entry             |
|      0.2        970128      fpl  Added dcbt logic             |
|      0.3        970203      fpl  Corrected ABIT define        |
|      0.4        970522      jfk  Added new dcbx test macros   |
|      0.5        980325      fpl  Added 740 code segment       |
|      0.6        980404      fpl  Removed loop stall           |
|      0.7        980708      fpl  Added build macros           |
|      0.8        980820      jfk  Added new DCBT macro         |
|      0.9        981019      fpl  Added z function             |
|      0.10       981025      fpl  Modified z entry             |
|      0.11       990310      fpl  750/G4 integration           |
|      0.12       990730      fpl  Added conjugate entry        |
|      1.0        000223      fpl  Increased minimum VMX count  |
|      1.1        000305      jfkremoved branches to entrypoints|
|      1.2        000607      jfk Fixed floating point save bug |
|      1.3        000610      fpl  Added new API macro          |
+--------------------------------------------------------------*/

#include "salppc.inc"

#undef BR IF VMX Z2
#define BR_IF_VMX_Z2( root_name, uroot name, min n imm, unit_s_imm, \
                      pr1, pi1, s1, pr2, pi2, s2, n, eflag ) \
   cmplwi n, min n_imm; \
   blt z_skip vmx; \
   cmpwi s1, unit s imm; \
   bne z_skip vmx; \
   cmpwi s2, unit s imm; \
   xor r0, pr1, pi1; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr2, pi2; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr2; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   bne z unaligned vmx; \
   BR VMX Z2( root_name, eflag, s1 ) \
z_unaligned vmx: \
   BR VMX Z2( uroot_name, eflag, s1 ) \
z_skip_vmx:

#define ACOND 5
#define ABIT  2
#define BCOND 6
```

fixed_cdotpr.mac 2/23/2001

```
#define BBIT  1

/**
 API registers
**/
#define A   r3
#define I   r4
#define B   r5
#define J   r6
#define C   r7
#define N   r8
#define EFLAG r9

/**
 z input args
**/
#define Ar  A
#define Ai  r10
#define Br  B
#define Bi  r11
#define Cr  C
#define Ci  r12

/**
 Local registers
**/
#define count r13
#define rtmp r13
#define nextline r14

/**
 Fpu registers
**/
#define rsumr0 f0
#define rsumi0 f1
#define isumr0 f2
#define isumi0 f3

#define ar0 f4
#define ai0 f5
#define ar1 f6
#define ai1 f7
#define ar2 f8
#define ai2 f9
#define ar3 f10
#define ai3 f11

#define br0 f12
#define bi0 f13
#define br1 f14
#define bi1 f15
#define br2 f16
#define bi2 f17
#define br3 f18
#define bi3 f19

#if defined( BUILD_MAX )
#if MCOS 55
DECLARE_VMX_Z2( _zdotpr_vmx_cc )
#else
DECLARE_VMX_Z2( _zdotpr_vmx )
#endif
DECLARE_VMX_Z2( _zdotpr4_vmx )
#endif

/**
 Code text: Conjugate
```

fixed_cdotpr.mac 2/23/2001

```
**/
FUNC PROLOG
#ifndef COMPILE C
U_ENTRY( fixed cidotpr  )                    /* Fortran SAL */
   FORTRAN DREF 3( I, J, N )
U_ENTRY( fixed cidotpr )                     /* C SAL */
   LI( EFLAG, SAL NNN )                /* NNN EFLAG (default) */
   BR( cidotprx common )               /* common path */
U_ENTRY( fixed cidotprx  )                   /* Fortran ESAL */
   FORTRAN DREF 4( I, J, N, EFLAG )
U ENTRY( fixed cidotprx )             /* C ESAL */
LABEL( cidotprx common )               /* common path */
   ADDI( Ai, Ar, 4 )
   MR( Bi, Br )
   ADDI( Br, Br, 4 )
   MR( Ci, Cr )
   ADDI( Cr, Cr, 4 )
   BR( common )                /* common path */
/**
 Normal
**/
FUNC PROLOG
#ifndef COMPILE C
U_ENTRY( fixed cdotpr_ )                    /* Fortran SAL */
   FORTRAN DREF 3( I, J, N )
U_ENTRY( fixed cdotpr )                     /* C SAL */
   LI( EFLAG, SAL NNN )               /* NNN EFLAG (default) */
   BR( cdotprx common )               /* common path */
U_ENTRY( fixed cdotprx  )                   /* Fortran ESAL */
   FORTRAN DREF 4( I, J, N, EFLAG )
U ENTRY( fixed cdotprx )             /* C ESAL */
LABEL( cdotprx common )               /* common path */
   ADDI( Ai, Ar, 4 )
   ADDI( Bi, Br, 4 )
   ADDI( Ci, Cr, 4 )
   BR( common )                /* common path */
/**
 Split complex entries: Conjugate
**/
U_ENTRY( fixed zidotpr  )             /* Fortran SAL */
   FORTRAN DREF 3( I, J, N )
U_ENTRY( fixed zidotpr )              /* C SAL */
   LI( EFLAG, SAL NNN )         /* NNN EFLAG (default) */
   BR( zidotprx common )
U_ENTRY( fixed zidotprx  )            /* Fortran ESAL */
   FORTRAN_DREF_4( I, J, N, EFLAG )
#endif
ENTRY 7( fixed zidotprx, A, I, B, J, C, N, EFLAG)
LABEL( zidotprx_common )
/**
 Assign split complex pointers, do the conjugate trick
**/
   LWZ( Ai, A, 4 )
   LWZ( Ar, A, 0 )
   LWZ( Bi, B, 0 )
   LWZ( Br, B, 4 )
   LWZ( Ci, C, 0 )
   LWZ( Cr, C, 4 )
   BR( z_common )
/**
 Normal
**/
U_ENTRY( fixed zdotpr_ )             /* Fortran SAL */
   FORTRAN DREF 3( I, J, N )
U_ENTRY( fixed zdotpr )              /* C SAL */
   LI( EFLAG, SAL NNN )        /* NNN EFLAG (default) */
   BR( zdotprx_common )
```

fixed_cdotpr.mac 2/23/2001

```
U_ENTRY( fixed zdotprx  )          /* Fortran ESAL */
   FORTRAN_DREF_4( I, J, N, EFLAG )
#endif
/**
 C ESAL
**/
ENTRY 7( fixed zdotprx, A, I, B, J, C, N, EFLAG)
   DECLARE r10 r14
   DECLARE_f0_f19

LABEL( zdotprx_common )

/**
 Assign split complex pointers
**/
   LWZ( Ai, A, 4 ) /* must load imag first since Ar reg = A reg */
   LWZ( Ar, A, 0 )
   LWZ( Bi, B, 4 )
   LWZ( Br, B, 0 )
   LWZ( Ci, C, 4 )
   LWZ( Cr, C, 0 )
/**
 VMX API filter
 Test if okay to enter VMX code and branch to VMX code
 VMX loop - process all N points
**/

LABEL( z_common )

#if defined( BUILD_MAX )

#define  MIN VMX N  20
#define  UNIT_STRIDE  1

#if MCOS 55
   BR_IF_VMX_Z2(  zdotpr_vmx cc,  zdotpr4_vmx, MIN_VMX_N, UNIT_STRIDE, \
                 Ar, Ai, I, Br, Bi, J, N, EFLAG )
#else
   BR_IF_VMX_Z2(  zdotpr_vmx,  zdotpr4_vmx, MIN_VMX_N, UNIT_STRIDE, \
                 Ar, Ai, I, Br, Bi, J, N, EFLAG )
#endif

#endif  /* BUILD_MAX */

/**
 Point of common path where all entries join
 Test for small counts
**/
LABEL( common )
   SAVE r13 r14
   SAVE f14 f19
   CMPLWI(N, 0)
   BEQ(ret)
   CMPLWI(N, 1)
   BEQ(do1)
   CMPLWI(N, 2)
   BEQ(do2)
   CMPLWI(N, 3)
   BEQ(do3)
/**
 check for uncached (and local) vectors
**/
   SET_2_DCBT_COND( ACOND, ABIT, BCOND, BBIT, EFLAG, rtmp )

   LI(nextline, 32)
/**
 740 code segment, start up loop code
```

fixed_cdotpr.mac 2/23/2001

```
**/
#if defined( BUILD 750 ) || defined( BUILD_MAX )
    LFS( ar0, Ar, 0 )
    SRWI( count, N, 2 ) /* count = N >> 2 */
    LFS( br0, Br, 0 )
    SLWI( I, I, 2 )      /* byte strides */
    LFS( ai0, Ai, 0 )
    SLWI( J, J, 2 )
    LFS( bi0, Bi, 0 )

    LFSUX( ar1, Ar, I )
    LFSUX( br1, Br, J )
    LFSUX( ai1, Ai, I )
    LFSUX( bi1, Bi, J )

    LFSUX( ar2, Ar, I )
    LFSUX( br2, Br, J )
    LFSUX( ai2, Ai, I )
    LFSUX( bi2, Bi, J )

    FMULS( rsumr0, ar0, br0 )
    LFSUX( ar3, Ar, I )
    LFSUX( br3, Br, J )
    FMULS( rsumi0, ai0, bi0 )
    LFSUX( ai3, Ai, I )
    LFSUX( bi3, Bi, J )
    FMULS( isumi0, ar0, bi0 )
    DECR C( count )
    FMULS( isumr0, ai0, br0 )
    BEQ(flush loop_740)
    BR(mloop_740)
/**
 Top of 740 loop
**/
LABEL(loop_740)
    LFSUX( ar3, Ar, I )
    FMADDS( rsumr0, ar0, br0, rsumr0 )
    LFSUX( br3, Br, J )
    FMADDS( rsumi0, ai0, bi0, rsumi0 )
    LFSUX( ai3, Ai, I )
    FMADDS( isumi0, ar0, bi0, isumi0 )
    FMADDS( isumr0, ai0, br0, isumr0 )
    LFSUX( bi3, Bi, J )

LABEL(mloop_740)
    FMADDS( rsumr0, ar1, br1, rsumr0 )
    LFSUX( ar0, Ar, I )

    DCBT IF( ACOND, Ar, nextline )
    FMADDS( rsumi0, ai1, bi1, rsumi0 )
    LFSUX( br0, Br, J )

    DECR C( count )
    FMADDS( isumi0, ar1, bi1, isumi0 )
    LFSUX( ai0, Ai, I )
    FMADDS( isumr0, ai1, br1, isumr0 )
    LFSUX( bi0, Bi, J )

    DCBT IF( BCOND, Br, nextline )
    FMADDS( rsumr0, ar2, br2, rsumr0 )
    LFSUX( ar1, Ar, I )
    LFSUX( br1, Br, J )
    FMADDS( rsumi0, ai2, bi2, rsumi0 )
    LFSUX( ai1, Ai, I )
    FMADDS( isumi0, ar2, bi2, isumi0 )
    LFSUX( bi1, Bi, J )
    FMADDS( isumr0, ai2, br2, isumr0 )
```

```
    FMADDS( rsumr0, ar3, br3, rsumr0 )
    LFSUX( ar2, Ar, I )
    FMADDS( rsumi0, ai3, bi3, rsumi0 )
    LFSUX( br2, Br, J )
    FMADDS( isumi0, ar3, bi3, isumi0 )
    LFSUX( ai2, Ai, I )
    LFSUX( bi2, Bi, J )
    FMADDS( isumr0, ai3, br3, isumr0 )
    BNE( loop_740 )
/**
 Finish last pass
**/
    FMADDS( rsumr0, ar0, br0, rsumr0 )
    LFSUX( ar3, Ar, I )
    LFSUX( br3, Br, J )
    FMADDS( rsumi0, ai0, bi0, rsumi0 )
    LFSUX( ai3, Ai, I )
    LFSUX( bi3, Bi, J )
    FMADDS( isumi0, ar0, bi0, isumi0 )
    FMADDS( isumr0, ai0, br0, isumr0 )

LABEL( flush loop 740 )
    FMADDS( rsumr0, ar1, br1, rsumr0 )
    FMADDS( rsumi0, ai1, bi1, rsumi0 )

    FMADDS( isumi0, ar1, bi1, isumi0 )
    FMADDS( isumr0, ai1, br1, isumr0 )

    FMADDS( rsumr0, ar2, br2, rsumr0 )
    FMADDS( rsumi0, ai2, bi2, rsumi0 )
    FMADDS( isumi0, ar2, bi2, isumi0 )
    FMADDS( isumr0, ai2, br2, isumr0 )

    FMADDS( rsumr0, ar3, br3, rsumr0 )
    FMADDS( rsumi0, ai3, bi3, rsumi0 )
    FMADDS( isumi0, ar3, bi3, isumi0 )
    FMADDS( isumr0, ai3, br3, isumr0 )
    BR(remain)
#endif   /** 750 specific code section **/
/**
 set up for loop entry, here if N >= 2
**/
#if defined( BUILD_603 )
LABEL(start 603)
   LFS( ar0, Ar, 0 )
   SLWI( I, I, 2 )      /* byte strides */
   LFS( ai0, Ai, 0 )
   SRWI( count, N, 2 ) /* count = N >> 2 */
   LFSUX( ar1, Ar, I )
   SLWI( J, J, 2 )
   LFSUX( ai1, Ai, I )
   LFSUX( ar2, Ar, I )
   LFSUX( ai2, Ai, I )
   LFSUX( ar3, Ar, I )
   LFSUX( ai3, Ai, I )
   DCBT_IF( ACOND, Ar, nextline )

   LFS( br0, Br, 0 )
   DECR_C( count )
   LFS( bi0, Bi, 0 )
   LFSUX( br1, Br, J )
   LFSUX( bi1, Bi, J )
   LFSUX( br2, Br, J )
   LFSUX( bi2, Bi, J )
   LFSUX( br3, Br, J )
   LFSUX( bi3, Bi, J )
```

fixed_cdotpr.mac 2/23/2001

```
   DCBT_IF( BCOND, Br, nextline )

   FMULS( rsumr0, ar0, br0 )
   FMULS( rsumi0, ai0, bi0 )
   FMULS( isumi0, ar0, bi0 )
   FMULS( isumr0, ai0, br0 )

   FMADDS( rsumr0, ar1, br1, rsumr0 )
   FMADDS( rsumi0, ai1, bi1, rsumi0 )
   FMADDS( isumi0, ar1, bi1, isumi0 )
   FMADDS( isumr0, ai1, br1, isumr0 )

   FMADDS( rsumr0, ar2, br2, rsumr0 )
   FMADDS( rsumi0, ai2, bi2, rsumi0 )
   FMADDS( isumi0, ar2, bi2, isumi0 )
   FMADDS( isumr0, ai2, br2, isumr0 )

   FMADDS( rsumr0, ar3, br3, rsumr0 )
   FMADDS( rsumi0, ai3, bi3, rsumi0 )
   FMADDS( isumi0, ar3, bi3, isumi0 )
   FMADDS( isumr0, ai3, br3, isumr0 )
   BEQ( remain )
/**
 main loop maintains four partial sums
 representing two complex sum updates per pass
**/
LABEL(loop)
    LFSUX( ar0, Ar, I )
    LFSUX( ai0, Ai, I )
    LFSUX( ar1, Ar, I )
    LFSUX( ai1, Ai, I )
    LFSUX( ar2, Ar, I )
    LFSUX( ai2, Ai, I )
    LFSUX( ar3, Ar, I )
    LFSUX( ai3, Ai, I )
    DCBT_IF( ACOND, Ar, nextline )

    DECR C( count )
    LFSUX( br0, Br, J )
    LFSUX( bi0, Bi, J )
    LFSUX( br1, Br, J )
    LFSUX( bi1, Bi, J )
    LFSUX( br2, Br, J )
    LFSUX( bi2, Bi, J )
    LFSUX( br3, Br, J )
    LFSUX( bi3, Bi, J )
    DCBT_IF( BCOND, Br, nextline )

    FMADDS( rsumr0, ar0, br0, rsumr0 )
    FMADDS( rsumi0, ai0, bi0, rsumi0 )
    FMADDS( isumi0, ar0, bi0, isumi0 )
    FMADDS( isumr0, ai0, br0, isumr0 )

    FMADDS( rsumr0, ar1, br1, rsumr0 )
    FMADDS( rsumi0, ai1, bi1, rsumi0 )
    FMADDS( isumi0, ar1, bi1, isumi0 )
    FMADDS( isumr0, ai1, br1, isumr0 )

    FMADDS( rsumr0, ar2, br2, rsumr0 )
    FMADDS( rsumi0, ai2, bi2, rsumi0 )
    FMADDS( isumi0, ar2, bi2, isumi0 )
    FMADDS( isumr0, ai2, br2, isumr0 )

    FMADDS( rsumr0, ar3, br3, rsumr0 )
    FMADDS( rsumi0, ai3, bi3, rsumi0 )
    FMADDS( isumi0, ar3, bi3, isumi0 )
    FMADDS( isumr0, ai3, br3, isumr0 )
```

fixed_cdotpr.mac 2/23/2001

```
    BNE( loop )
#endif   /** 603 specific code section **/
/**
 remainder loop
**/
LABEL(remain)
   ANDI_C( count, N, 2 )  /* bit 2 */
   BEQ( sum1 )

   LFSUX( ar0, Ar, I )
   LFSUX( ai0, Ai, I )
   LFSUX( ar1, Ar, I )
   LFSUX( ai1, Ai, I )

   LFSUX( br0, Br, J )
   LFSUX( bi0, Bi, J )
   LFSUX( br1, Br, J )
   LFSUX( bi1, Bi, J )

   FMADDS( rsumr0, ar0, br0, rsumr0 )
   FMADDS( rsumi0, ai0, bi0, rsumi0 )
   FMADDS( isumi0, ar0, bi0, isumi0 )
   FMADDS( isumr0, ai0, br0, isumr0 )

   FMADDS( rsumr0, ar1, br1, rsumr0 )
   FMADDS( rsumi0, ai1, bi1, rsumi0 )
   FMADDS( isumi0, ar1, bi1, isumi0 )
   FMADDS( isumr0, ai1, br1, isumr0 )

LABEL(sum1)
   ANDI_C( count, N, 1 )  /* bit 0 */
   BEQ( combine )         /* if no sums left */

   LFSUX( ar0, Ar, I )
   LFSUX( br0, Br, J )
   LFSUX( ai0, Ai, I )
   LFSUX( bi0, Bi, J )

   FMADDS( rsumr0, ar0, br0, rsumr0 )
   FMADDS( rsumi0, ai0, bi0, rsumi0 )
   FMADDS( isumi0, ar0, bi0, isumi0 )
   FMADDS( isumr0, ai0, br0, isumr0 )
/**
 combine partial sums, write out results and return
**/
LABEL(combine)
   FSUBS( rsumr0, rsumr0, rsumi0 ) /** rsumr0 = rsumr0 - rsumi0 **/
   STFS( rsumr0, Cr, 0 )           /** *(S + 0) = rsumr0 **/
   FADDS( isumi0, isumi0, isumr0 )
   STFS( isumi0, Ci, 0 )
   BR(ret)
/**
 here for N = 1,2,3
**/
LABEL(do3)
   LFS( ar0, Ar, 0 )
   SLWI( I, I, 2 )     /* byte strides */
   LFS( ai0, Ai, 0 )
   LFSUX( ar1, Ar, I )
   SLWI( J, J, 2 )
   LFSUX( ai1, Ai, I )
   LFSUX( ar2, Ar, I )
   LFSUX( ai2, Ai, I )

   LFS( br0, Br, 0 )
   DECR_C( count )
   LFS( bi0, Bi, 0 )
```

```
   LFSUX( br1, Br, J )
   LFSUX( bi1, Bi, J )
   LFSUX( br2, Br, J )
   LFSUX( bi2, Bi, J )

   FMULS( rsumr0, ar0, br0 )
   FMULS( rsumi0, ai0, bi0 )
   FMULS( isumi0, ar0, bi0 )
   FMULS( isumr0, ai0, br0 )

   FMADDS( rsumr0, ar1, br1, rsumr0 )
   FMADDS( rsumi0, ai1, bi1, rsumi0 )
   FMADDS( isumi0, ar1, bi1, isumi0 )
   FMADDS( isumr0, ai1, br1, isumr0 )

   FMADDS( rsumr0, ar2, br2, rsumr0 )
   FMADDS( rsumi0, ai2, bi2, rsumi0 )
   FMADDS( isumi0, ar2, bi2, isumi0 )
   FMADDS( isumr0, ai2, br2, isumr0 )
   BR(combine)

LABEL(do2)
   LFS( ar0, Ar, 0 )
   SLWI( I, I, 2 )      /* byte strides */
   LFS( ai0, Ai, 0 )
   LFSUX( ar1, Ar, I )
   SLWI( J, J, 2 )
   LFSUX( ai1, Ai, I )

   LFS( br0, Br, 0 )
   LFS( bi0, Bi, 0 )
   LFSUX( br1, Br, J )
   LFSUX( bi1, Bi, J )

   FMULS( rsumr0, ar0, br0 )
   FMULS( rsumi0, ai0, bi0 )
   FMULS( isumi0, ar0, bi0 )
   FMULS( isumr0, ai0, br0 )

   FMADDS( rsumr0, ar1, br1, rsumr0 )
   FMADDS( rsumi0, ai1, bi1, rsumi0 )
   FMADDS( isumi0, ar1, bi1, isumi0 )
   FMADDS( isumr0, ai1, br1, isumr0 )
   BR(combine)

LABEL(do1)
   LFS( ai0, Ai, 0 )
   LFS( bi0, Bi, 0 )
   LFS( br0, Br, 0 )
   LFS( ar0, Ar, 0 )

   FMULS( rsumi0, ai0, bi0)
   FMULS( isumr0, ai0, br0)
   FMSUBS( rsumr0, ar0, br0, rsumi0)
   STFS( rsumr0, Cr, 0 )
   FMADDS( isumi0, ar0, bi0, isumr0)
   STFS( isumi0, Ci, 0 )
/**
 return
**/
LABEL(ret)
   REST f14 f19
   REST r13_r14
   RETURN
FUNC_EPILOG
```

gen_r_sums.mac 2/23/2001

```
/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version   ---|
|--------------------------------------------------------------|
|                                                              |
|  File Name:    GEN R SUMS.MAC                                |
|  Description:  Multiple small dot product routine for wireless|
|                group application.                            |
|  Entry/params:                                               |
|  GEN_R_SUMS (X_bf, Coor_bf, Ptov_map, R_sums, Num_phys_users) |
|                                                              |
|  Formula:                                                    |
|    num_sums = 0;                                             |
|    for ( i = 0; i < Num phys users; i++ ) {                  |
|      for ( j = 0; j < (int)Ptov_map[i]; j++ ) {              |
|        sum = 0;                                              |
|        for ( k = 0; k < 16; k++ ) {                          |
|          sum += (BF32)X bf[k].real * (BF32)Corr bf->real;    |
|          sum += (BF32)X_bf[k].imag * (BF32)Corr_bf->imag;    |
|          ++Corr_bf;                                          |
|        }                                                     |
|        *R sums++ = sum;                                      |
|        ++num_sums;                                           |
|      }                                                       |
|      X_bf += N_FINGERS_MAX_SQUARED;                          |
|    }                                                         |
|             Mercury Computer Systems, Inc.                   |
|             Copyright (c) 2000  All rights reserved          |
|                                                              |
|   Revision       Date      Engineer  Reason                   |
|   --------       ----      --------  --------                 |
|     0.0         000906       fpl    Created                   |
+-------------------------------------------------------------*/

#include "salppc.inc"

#define DO IO  1
#define DO_PREFETCH  1

#if DO IO
#define  PTOV BUMP 1  1
#define  CORR BUMP 32  32
#define  CORR BUMP_64  64
#define  X BUMP 64  64
#define  RSUM BUMP 8  8
#define  RSUM_BUMP_4  4
#else
#define  PTOV BUMP 1  0
#define  CORR BUMP 32  0
#define  CORR BUMP_64  0
#define  X BUMP 64  0
#define  RSUM BUMP 8  0
#define  RSUM_BUMP_4  0
#endif

#define LOAD_CORR( vT, rA, rB )   LVX( vT, rA, rB )

#define  DST_BUMP  CORR_BUMP_64

#if DO PREFETCH
#define  PREFETCH( rA, rB, STRM ) \
   DST( rA, rB, STRM ) \
   ADDI( rA, rA, DST_BUMP )
#else
#define  PREFETCH( rA, rB, STRM )
#endif
```

```
gen_r_sums.mac                                                    2/23/2001

#define OLOOP_BIT 6

/**
 Input parameters
**/
#define X bf             r3
#define Corr bf          r4
#define Ptov map         r5
#define R sump           r6
#define Num_phys_users r7
/**
 Local GPRs
**/
#define icount r8
#define ptov count r9
#define indx1 r10
#define indx2 r11
#define indx3 r12
#define sindex1 r13
#define dstp r14
#define dst_code r15
/**
 G4 registers
**/
#define corr00 v0
#define corr01 v1
#define corr10 v2
#define corr11 v3

#define C0 0 v4
#define C1 0 v5
#define C0 8 v6
#define C1_8 v7

#define C0 16 v8
#define C1 16 v9
#define C0 24 v10
#define C1_24 v11

#define X0   v12
#define X8   v13
#define X16  v14
#define X24  v15

#define sum0 v16
#define sum1 v17
#define zero v18

/**
 Begin code text
**/
FUNC PROLOG
ENTRY_5( gen_R_sums, X_bf, Corr_bf, Ptov_map, R_sump, Num_phys_users )

  CMPWI( Num_phys_users, 0 )
  BGT( start )
  RETURN

LABEL( start )

  SAVE r13 r15
  USE_THRU_v18( VRSAVE_COND )

/**
 DST setup
**/
  MAKE_STREAM_CODE_IIR( dst_code, DST_BUMP, 1, 0 )
```

```
  ADDI( dstp, Corr bf, 80 )            /* start prefetch advanced */
  PREFETCH( dstp, dst_code, 0 )

/**
 Setup for outer loop entry
 Read and expand two coor vectors
 Set outer loop counter condition
**/
  LI( indx1, 16 )
  LI( indx2, 32 )
  LI( indx3, 48 )
  LI( sindex1, 4 )
  CMPWI CR( OLOOP_BIT, Num phys_users, 0 )
  LVX( corr00, 0, Corr bf )
  VXOR( zero, zero, zero )
  LVX( corr01, Corr bf, indx1 )
  LVX( corr10, Corr bf, indx2 )
  LVX( corr11, Corr_bf, indx3 )

  VUPKHSB( C0 0, corr00 )
  ADDI( Corr bf, Corr bf, CORR_BUMP_64 )
  VUPKLSB( C0 8, corr00 )
  ADDI( Ptov map, Ptov map, -PTOV_BUMP_1 )
  VUPKHSB( C1 0, corr10 )
  ADDI( R sump, R sump, -RSUM_BUMP_8 )
  VUPKLSB( C1 8, corr10 )
  VUPKHSB( C0 16, corr01 )
  VUPKLSB( C0 24, corr01 )
  VUPKHSB( C1 16, corr11 )
  VUPKLSB( C1_24, corr11 )
/**
 Outer loop for each physical user
**/
LABEL( oloop )
/* { */
   DECR( Num phys users )
   LBZU( ptov count, Ptov_map, 1 )
   BEQ CR( OLOOP BIT, ret )
   LVX( X0, 0, X bf )
   LVX( X8, X bf, indx1 )
   SRWI_C( icount, ptov count, 1 )
   LVX( X16, X bf, indx2 )
   LVX( X24, X_bf, indx3 )
   ADDI( X bf, X bf, X BUMP 64 )
   CMPWI CR( OLOOP BIT, Num_phys_users, 0 )
   BEQ_MINUS( one_sum )
/**
 Top of sum loop
 Produces two sums each pass
**/
LABEL( iloop )
/* { */
    PREFETCH( dstp, dst code, 0 )
    VMSUMSHS( sum0, C0 0, X0, zero )
    VMSUMSHS( sum1, C1 0, X0, zero )
    LVX( corr00, 0, Corr bf )
    LVX( corr01, Corr bf, indx1 )
    LVX( corr10, Corr bf, indx2 )
    VMSUMSHS( sum0, C0_8, X8, sum0 )
    DECR C( icount )
    VMSUMSHS( sum1, C1 8, X8, sum1 )
    LVX( corr11, Corr bf, indx3 )
    VUPKHSB( C0 0, corr00 )
    VUPKLSB( C0 8, corr00 )
    VMSUMSHS( sum0, C0 16, X16, sum0 )
    VMSUMSHS( sum1, C1 16, X16, sum1 )
    VUPKHSB( C1_0, corr10 )
```

```
    ADDI( R sump, R sump, RSUM_BUMP_8 )
    VUPKLSB( C1 8, corr10 )
    VMSUMSHS( sum0, C0 24, X24, sum0 )
    VUPKHSB( C0 16, corr01 )
    VMSUMSHS( sum1, C1 24, X24, sum1 )
    VUPKLSB( C0 24, corr01 )
    VUPKHSB( C1 16, corr11 )
    VSUMSWS( sum0, sum0, zero )
    VUPKLSB( C1 24, corr11 )
    VSUMSWS( sum1, sum1, zero )
    ADDI( Corr bf, Corr_bf, CORR_BUMP_64 )
    VSPLTW( sum0, sum0, 3 )
    STVEWX( sum0, 0, R sump )
    VSPLTW( sum1, sum1, 3 )
    STVEWX( sum1, R_sump, sindex1 )
/* } */
  BNE( iloop )
/**
 Drop out, check for remainders
**/
   ANDI_C(icount, ptov_count, 0x1)
   BEQ( oloop )
/**
 One more sum:
 Enters and exits with two coor vectors are loaded and expanded to 16 bit
**/
LABEL( one sum )
   VMSUMSHS( sum0, C0 0, X0, zero )
   VMSUMSHS( sum0, C0 8, X8, sum0 )
   ADDI( R sump, R_sump, RSUM BUMP 8 )
   VMSUMSHS( sum0, C0 16, X16, sum0 )
   VMSUMSHS( sum0, C0 24, X24, sum0 )
   VSUMSWS( sum0, sum0, zero )

   VSPLTW( sum0, sum0, 3 )
   STVEWX( sum0, 0, R sump )
   ADDI( R_sump, R_sump, -RSUM_BUMP_4 )  /* pre-dec pointer for loop reentry
   */
/**
 Seup for loop re-entry: corr00 consumed in one_sum section
   loop exit ptr v
 corr00  corr10  corr00  corr10
         corr00  corr10  corr00  corr10
       loop re-entry ptr ^
**/
   VMR( corr00, corr10 )
   LVX( corr10, 0, Corr bf )
   VMR( corr01, corr11 )
   LVX( corr11, Corr bf, indx1 )
   ADDI( Corr_bf, Corr_bf, CORR_BUMP_32 )

   VUPKHSB( C0 0, corr00 )
   VUPKLSB( C0 8, corr00 )
   VUPKHSB( C1 0, corr10 )
   VUPKLSB( C1 8, corr10 )
   VUPKHSB( C0 16, corr01 )
   VUPKLSB( C0 24, corr01 )
   VUPKHSB( C1 16, corr11 )
   VUPKLSB( C1_24, corr11 )
/* } */
  BR( oloop )
/**
 Exit routine
**/
LABEL( ret )
  FREE THRU v18( VRSAVE_COND )
  REST_r13_r15
```

```
  RETURN
FUNC_EPILOG
```

gen_r_sums2.mac 2/23/2001

```
/*-----------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version    ---|
|----------------------------------------------------------------|
|                                                                |
|  File Name:     GEN R SUMS2.MAC                                |
|  Description:   Multiple small dot product routine for wireless|
|                 group application.                             |
|  Entry/params:   GEN R SUMS2 (X bf, Corr0 bf, Corr1 bf,        |
|                    Ptov_map, R_sums0, R_sums1, Num_phys_users) |
|                                                                |
|  Formula:                                                      |
|    num_sums = 0;                                               |
|    for ( i = 0; i < Num phys users; i++ ) {                    |
|      for ( j = 0; j < (int)Ptov_map[i]; j++ ) {                |
|        sum = 0;                                                |
|        for ( k = 0; k < 16; k++ ) {                            |
|          sum0 += (BF32)X bf[k].real * (BF32)Corr0 bf->real;    |
|          sum0 += (BF32)X_bf[k].imag * (BF32)Corr0_bf->imag;    |
|                                                                |
|          sum1 += (BF32)X bf[k].real * (BF32)Corr1 bf->real;    |
|          sum1 += (BF32)X_bf[k].imag * (BF32)Corr1_bf->imag;    |
|          ++Corr0 bf;                                           |
|          ++Corr1_bf;                                           |
|        }                                                       |
|                                                                |
|        *R sums0++ = sum0;                                      |
|        *R sums1++ = sum1;                                      |
|        ++num_sums;                                             |
|      }                                                         |
|      X_bf += N_FINGERS_MAX_SQUARED;                            |
|    }                                                           |
|              Mercury Computer Systems, Inc.                    |
|              Copyright (c) 2000  All rights reserved           |
|                                                                |
|   Revision      Date      Engineer  Reason                     |
|   --------      ----      --------  --------                   |
|     0.0        000906       fpl     Created                    |
|     0.1        000908       fpl     Fixed zero bug             |
+---------------------------------------------------------------*/

#include "salppc.inc"

#define DO IO  1
#define DO_PREFETCH  1

#if DO IO
#define   PTOV BUMP 1  1
#define   CORR BUMP 32  32
#define   CORR BUMP_64  64
#define   X BUMP 64  64
#define   RSUM BUMP 8  8
#define   RSUM_BUMP_4  4
#else
#define   PTOV BUMP 1  0
#define   CORR BUMP 32  0
#define   CORR BUMP_64  0
#define   X BUMP 64  0
#define   RSUM BUMP 8  0
#define   RSUM_BUMP_4  0
#endif

#define LOAD_CORR( vT, rA, rB )   LVX( vT, rA, rB )

#define   DST_BUMP   CORR_BUMP_64

#if DO PREFETCH
#define   PREFETCH( rA, rB, STRM ) \
```

gen_r_sums2.mac 2/23/2001

```
   DST( rA, rB, STRM ) \
   ADDI( rA, rA, DST_BUMP )
#else
#define  PREFETCH( rA, rB, STRM )
#endif

#define OLOOP_BIT 6

/**
 Input parameters
**/
#define X bf             r3
#define Corr0 bf         r4
#define Corr1 bf         r5
#define Ptov map         r6
#define R sump0          r7
#define R sump1          r8
#define Num_phys_users r9
/**
 Local GPRs
**/
#define icount r10
#define ptov count r11
#define indx1 r12
#define indx2 r13
#define indx3 r14
#define sindex1 r15
#define dstp r16
#define dst code r17
#define dst_stride  indx3
/**
 G4 registers
**/
#define corr00 v0
#define corr01 v1
#define corr10 v2
#define corr11 v3
#define corr20 v4
#define corr21 v5
#define corr30 v6
#define corr31 corr00
#define zero   v7

#define C0 0 v8
#define C1 0 v9
#define C2 0 v10
#define C3_0 v11

#define C0 8 v12
#define C1 8 v13
#define C2 8 v14
#define C3_8 v15

#define C0 16 v16
#define C1 16 v17
#define C2 16 v18
#define C3_16 v19

#define C0 24 v20
#define C1 24 v21
#define C2 24 v22
#define C3_24 v23

#define X0  v24
#define X8  v25
#define X16 v26
#define X24 v27
```

gen_r_sums2.mac 2/23/2001

```
#define sum0 v28
#define sum1 v29
#define sum2 v30
#define sum3 v31
/**
 Begin code text
**/
FUNC_PROLOG

#if 1
NOP                          /****** alignment may be important ******/
#endif

ENTRY_7( gen R sums2, X_bf, Corr0_bf, Corr1_bf, Ptov_map, R_sump0, R_sump1,
        Num_phys_users )

  CMPWI( Num_phys_users, 0 )
  BGT( start )
  RETURN

LABEL( start )
  SAVE r13 r17
  USE_THRU_v31( VRSAVE_COND )
/**
 DST setup
**/
  SUB( dst stride, Corr1 bf, Corr0 bf )
  MAKE STREAM_CODE IIR( dst code, DST_BUMP, 2, dst_stride )
  ADDI( dstp, Corr0 bf, 80 )            /* start prefetch advanced */
      /* 48: 1087, 64: 1094, 80: 1043, 96: 1058, 112: 1049, 128: 1061 */
  PREFETCH( dstp, dst_code, 0 )

/**
 Setup for outer loop entry
 Read and expand two coor vectors
 Set outer loop counter condition
**/
  LI( indx1, 16 )
  LI( indx2, 32 )
  LI( indx3, 48 )
  LI( sindex1, 4 )
  CMPWI_CR( OLOOP_BIT, Num_phys_users, 0 )

  LOAD CORR( corr00, 0, Corr0 bf )
  LOAD CORR( corr10, Corr0 bf, indx2 )
  ADDI( Ptov_map, Ptov map, -PTOV BUMP_1 )
  LOAD CORR( corr20, 0, Corr1 bf )
  ADDI( R sump0, R sump0, -RSUM BUMP_8 )
  LOAD_CORR( corr30, Corr1_bf, indx2 )

  LOAD CORR( corr01, Corr0 bf, indx1 )
  ADDI( R sump1, R sump1, -RSUM BUMP_8 )
  LOAD CORR( corr11, Corr0_bf, indx3 )
  VXOR( zero, zero, zero )
  LOAD_CORR( corr21, Corr1_bf, indx1 )

  VUPKHSB( C0 0, corr00 )
  ADDI( Corr0 bf, Corr0_bf, CORR_BUMP_64 )
  VUPKHSB( C1 0, corr10 )
  VUPKHSB( C2 0, corr20 )
  VUPKHSB( C3_0, corr30 )

  VUPKLSB( C0 8, corr00 )
  LOAD CORR( corr31, Corr1_bf, indx3 ) /* corr00, corr31 same register */
  VUPKLSB( C1 8, corr10 )
  ADDI( Corr1_bf, Corr1_bf, CORR_BUMP_64 )
```

```
  VUPKLSB( C2 8, corr20 )
  VUPKLSB( C3_8, corr30 )

  VUPKHSB( C0 16, corr01 )
  VUPKHSB( C1 16, corr11 )
  VUPKHSB( C2 16, corr21 )
  VUPKHSB( C3_16, corr31 )

  VUPKLSB( C0 24, corr01 )
  VUPKLSB( C1 24, corr11 )
  VUPKLSB( C2 24, corr21 )
  VUPKLSB( C3_24, corr31 )
/**
 Outer loop for each physical user
**/
LABEL( oloop )
/* { */
   DECR( Num phys users )
   LBZU( ptov count, Ptov_map, PTOV_BUMP_1 )
   BEQ CR( OLOOP BIT, ret )
   LVX( X0, 0, X bf )
   LVX( X8, X bf, indx1 )
   SRWI_C( icount, ptov count, 1 )
   LVX( X16, X bf, indx2 )
   LVX( X24, X_bf, indx3 )
   ADDI( X bf, X bf, X BUMP 64 )
   CMPWI CR( OLOOP BIT, Num_phys_users, 0 )
   BEQ_MINUS( one_sum )
/**
 Top of sum loop
 Produces four sums each pass
**/
LABEL( iloop )
/* { */
    PREFETCH( dstp, dst code, 0 )
    LOAD CORR( corr00, 0, Corr0_bf )
    DECR C( icount )
    LOAD CORR( corr10, Corr0 bf, indx2 )
    VMSUMSHS( sum0, C0_0, X0, zero )
    LOAD CORR( corr20, 0, Corr1 bf )
    VMSUMSHS( sum1, C1_0, X0, zero )
    LOAD CORR( corr30, Corr1 bf, indx2 )
    LOAD CORR( corr01, Corr0 bf, indx1 )
    VMSUMSHS( sum2, C2_0, X0, zero )
    LOAD CORR( corr11, Corr0 bf, indx3 )
    LOAD CORR( corr21, Corr1 bf, indx1 )
    VMSUMSHS( sum3, C3 0, X0, zero )
    VUPKHSB( C0 0, corr00 )
    VMSUMSHS( sum0, C0 8, X8, sum0 )
    VUPKHSB( C1 0, corr10 )
    ADDI( R sump0, R sump0, RSUM_BUMP_8 )
    VUPKHSB( C2 0, corr20 )
    VMSUMSHS( sum1, C1 8, X8, sum1 )
    VUPKHSB( C3 0, corr30 )
    VMSUMSHS( sum2, C2 8, X8, sum2 )
    VUPKLSB( C0 8, corr00 )
    VMSUMSHS( sum3, C3 8, X8, sum3 )
    ADDI( Corr0 bf, Corr0 bf, CORR BUMP 64 )
    LOAD CORR( corr31, Corr1_bf, indx3 ) /* corr00, corr31 same register */
    VUPKLSB( C1 8, corr10 )
    VMSUMSHS( sum0, C0 16, X16, sum0 )
    VUPKLSB( C2 8, corr20 )
    VMSUMSHS( sum1, C1 16, X16, sum1 )
    VUPKLSB( C3 8, corr30 )
    ADDI( R sump1, R sump1, RSUM_BUMP_8 )
    VUPKHSB( C0 16, corr01 )
    VMSUMSHS( sum2, C2_16, X16, sum2 )
```

```
    VUPKHSB( C1 16, corr11 )
    VMSUMSHS( sum3, C3 16, X16, sum3 )
    VUPKHSB( C2 16, corr21 )
    VMSUMSHS( sum0, C0 24, X24, sum0 )
    ADDI( Corr1 bf, Corr1 bf, CORR BUMP_64 )
    VMSUMSHS( sum1, C1 24, X24, sum1 )
    VUPKHSB( C3 16, corr31 )
    VMSUMSHS( sum2, C2 24, X24, sum2 )
    VUPKLSB( C0 24, corr01 )
    VMSUMSHS( sum3, C3 24, X24, sum3 )
    VSUMSWS( sum0, sum0, zero )
    VUPKLSB( C1 24, corr11 )
    VSUMSWS( sum1, sum1, zero )
    VUPKLSB( C2 24, corr21 )
    VSUMSWS( sum2, sum2, zero )
    VUPKLSB( C3 24, corr31 )
    VSPLTW( sum0, sum0, 3 )
    VSUMSWS( sum3, sum3, zero )
    VSPLTW( sum1, sum1, 3 )
    STVEWX( sum0, 0, R sump0 )
    VSPLTW( sum2, sum2, 3 )
    STVEWX( sum1, R sump0, sindex1 )
    VSPLTW( sum3, sum3, 3 )
    STVEWX( sum2, 0, R sump1 )
    STVEWX( sum3, R_sump1, sindex1 )
/* } */
   BNE( iloop )

/**
 Drop out, check for remainders
**/
   ANDI_C(icount, ptov_count, 0x1)
   BEQ( oloop )
/**
 One more sum:
 Enters and exits with two coor vectors are loaded and expanded to 16 bit
**/
LABEL( one sum )
   VMSUMSHS( sum0, C0 0, X0, zero )
   ADDI( R sump0, R sump0, RSUM BUMP_8 )
   VMSUMSHS( sum2, C2 0, X0, zero )
   ADDI( R sump1, R sump1, RSUM BUMP_8 )
   VMSUMSHS( sum0, C0 8, X8, sum0 )
   VMSUMSHS( sum2, C2 8, X8, sum2 )
   VMSUMSHS( sum0, C0 16, X16, sum0 )
   VMSUMSHS( sum2, C2 16, X16, sum2 )
   VMSUMSHS( sum0, C0 24, X24, sum0 )
   VMSUMSHS( sum2, C2 24, X24, sum2 )
   VSUMSWS( sum0, sum0, zero )
   VSUMSWS( sum2, sum2, zero )

   VSPLTW( sum0, sum0, 3 )
   STVEWX( sum0, 0, R sump0 )
   VSPLTW( sum2, sum2, 3 )
   STVEWX( sum2, 0, R sump1 )
   ADDI( R_sump0, R_sump0, -RSUM_BUMP_4 )  /* pre-dec pointers for loop
   reentry */
   ADDI( R_sump1, R_sump1, -RSUM_BUMP_4 )
/**
 Setup for loop re-entry: corr00 consumed in one_sum section
        exit ptr v
 corr00  corr10  corr00  corr10
         corr00  corr10  corr00  corr10
            re-entry ptr ^
**/
   VMR( corr21, corr31 )  /* corr00, corr31 same register */
   VMR( corr00, corr10 )
```

gen_r_sums2.mac 2/23/2001

```
   LOAD_CORR( corr10, 0, Corr0_bf )
   VMR( corr01, corr11 )
   LOAD_CORR( corr11, Corr0_bf, indx1 )
   VMR( corr20, corr30 )
   LOAD_CORR( corr30, 0, Corr1_bf )

   VUPKHSB( C0_0, corr00 )
   VUPKLSB( C0_8, corr00 )
   LOAD_CORR( corr31, Corr1_bf, indx1 ) /* corr00, corr31 same register */
   VUPKHSB( C1_0, corr10 )
   VUPKLSB( C1_8, corr10 )
   VUPKHSB( C2_0, corr20 )
   VUPKLSB( C2_8, corr20 )
   VUPKHSB( C3_0, corr30 )
   VUPKLSB( C3_8, corr30 )

   VUPKHSB( C0_16, corr01 )
   ADDI( Corr0_bf, Corr0_bf, CORR_BUMP_32 )
   VUPKLSB( C0_24, corr01 )
   ADDI( Corr1_bf, Corr1_bf, CORR_BUMP_32 )
   VUPKHSB( C1_16, corr11 )
   VUPKLSB( C1_24, corr11 )
   VUPKHSB( C2_16, corr21 )
   VUPKLSB( C2_24, corr21 )
   VUPKHSB( C3_16, corr31 )
   VUPKLSB( C3_24, corr31 )
/* } */
  BR( oloop )
/**
 Exit routine
**/
LABEL( ret )
  FREE_THRU_v31( VRSAVE_COND )
  REST_r13_r17
  RETURN
FUNC_EPILOG
```

```
/*-------------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version       ---|
|-------------------------------------------------------------------|
|
|  File Name:     GEN X ROW.MAC
|  Description:   2 Complex scalers (4x1) 2 complex vectors (4xN)
|                 16 bit complex multiplication producing a 16
|                 bit complex vector of length 16*N.
|
|  Entry/params: GEN_X_ROW (A1, A2, C, Phys_index, N)
|
|  Formula:
|
| for ( i = 0; i < tot_phys_users; i++ ) {
|
|   in mpath1p = mpath1 bf + (i * N FINGERS MAX);
|   in_mpath2p = mpath2_bf + (i * N_FINGERS_MAX);
|
|   j = 0;
|   for ( q1 = 0; q1 < N_FINGERS_MAX; q1++ ) {
|
|     s1r = (BF32)out mpath1p[q1].real;
|     s1i = (BF32)out mpath1p[q1].imag;
|     s2r = (BF32)out mpath2p[q1].real;
|     s2i = (BF32)out_mpath2p[q1].imag;
|
|     for ( q = 0; q < N_FINGERS_MAX; q++ ) {
|
|       a1r = (BF32)in mpath1p[q].real;
|       a1i = (BF32)in mpath1p[q].imag;
|       a2r = (BF32)in mpath2p[q].real;
|       a2i = (BF32)in_mpath2p[q].imag;
|
|       cr = (a1r * s1r) + (a1i * s1i);
|       ci = (a1r * s1i) - (a1i * s1r);
|       cr += (a2r * s2r) + (a2i * s2i);
|       ci += (a2r * s2i) - (a2i * s2r);
|
|       X_bf[i * N_FINGERS_MAX_SQUARED + j].real
|                                       = (BF16)(cr >> 16);
|       X_bf[i * N_FINGERS_MAX_SQUARED + j].imag
|                                       = (BF16)(ci >> 16);
|       ++j;
|     }
|   }
| }
|             Mercury Computer Systems, Inc.
|             Copyright (c) 2000  All rights reserved
|
|   Revision       Date      Engineer  Reason
|   --------       ----      --------  --------
|     0.0         000907       fpl     Created
+-------------------------------------------------------------------*/

#include "salppc.inc"

#define LOG N FINGERS MAX  2
#define LOG ELEMENT_SIZE  2
#define INDEX_SHIFT  (LOG_N_FINGERS_MAX + LOG_ELEMENT_SIZE)

/**
 Local read-only Permute vector table
**/

RODATA_SECTION( 6 )
```

```
gen_x_row.mac                                                    2/23/2001

START_L_ARRAY( local_table )

L_PERMUTE_MASK( 0x02031011, 0x06071415, 0x0a0b1819, 0x0e0f1c1d )

/**
 32 -> 16 bit: select the 16 MSBs of each 32 bit field
**/
L_PERMUTE_MASK( 0x00011011, 0x04051415, 0x08091819, 0x0c0d1c1d )

END_ARRAY

/**
 API registers
**/
#define A1          r3
#define A2          r4
#define C           r5
#define Phys_index r6
#define N           r7
/**
 Integer loop registers
**/
#define Cp0    C
#define Cp1    r8
#define sptr1  r8
#define Cp2    r9
#define sptr2  r9
#define Cp3    r10
#define tptr   r10
#define cindex r11
#define aindex r12
#define index  r12

/**
 G4 registers
**/
#define cr00 v0
#define cr01 v1
#define cr02 v2
#define cr03 v3

#define vtmp0 v0
#define vtmp2 v2

#define ci00 v4
#define ci01 v5
#define ci02 v6
#define ci03 v7

#define sr00 v8
#define sr01 v9
#define sr02 v10
#define sr03 v11

#define si00 v12
#define si01 v13
#define si02 v14
#define si03 v15

#define sr10 v16
#define sr11 v17
#define sr12 v18
#define sr13 v19

#define si10 v20
#define si11 v21
#define si12 v22
```

```
gen_x_row.mac                                                    2/23/2001

#define si13 v23

#define c0 v24
#define c1 v24
#define c2 v25
#define c3 v26

#define a00 v27
#define a10 v27
#define a01 v28
#define a11 v29

#define sval v28
#define neg_sval v29

#define vc v30
#define zero v31

/**
 Begin code text
**/
FUNC PROLOG
ENTRY 5( gen X row, A1, A2, C, Phys_index, N )
  USE_THRU_v31( VRSAVE_COND )
/**
 Load up complex scaler
 sval = sr0 si0  sr1 si1  sr2 si2  sr3 si3
**/
   LA( tptr, local table, 0 )
   VXOR( zero, zero, zero )
   LI(index, 0)
/**
 Byte offset into 16 bit complex vector
**/
   SLWI( Phys index, Phys index, INDEX_SHIFT )
   ADD( sptr1, A1, Phys index )
   ADD( sptr2, A2, Phys_index )
/**
 Load up first scaler:
 if sval = sr0,si0 sr1,si1 sr2,si2 sr3,si3
         = s0 s1 s2 s3
**/
   LVX( sval, sptr1, index ) /* read 4 16 bit complex values */
   VSUBSHS( neg sval, zero, sval ) /* negate complex scaler values */
   VMRGHW(vtmp0, sval, sval)      /* vtmp0 = s0 s0 s1 s1 */
   VMRGLW(vtmp2, sval, sval)      /* vtmp2 = s2 s2 s3 s3 */
   VMRGHW(sr00, vtmp0, vtmp0) /* sr0 = s0 s0 s0 s0 */
   VMRGLW(sr01, vtmp0, vtmp0) /* sr1 = s1 s1 s1 s1 */
   VMRGHW(sr02, vtmp2, vtmp2) /* sr2 = s2 s2 s2 s2 */
   VMRGLW(sr03, vtmp2, vtmp2) /* sr3 = s3 s3 s3 s3 */
/**
  if neg sval = sr0,si0 sr1,si1 sr2,si2 sr3,si3
  after perm:
           = si0,-sr0 si1,-sr1 si2,-sr2 si3,-sr3
           = ns0 ns1 ns2 ns3
**/
   LVX( vc, tptr, index )
   VPERM( neg sval, sval, neg sval, vc ) /* si -sr */
   VMRGHW(vtmp0, neg sval, neg sval) /* vtmp0 = ns0 ns0 ns1 ns1 */
   VMRGLW(vtmp2, neg sval, neg sval) /* vtmp2 = ns2 ns2 ns3 ns3 */
   VMRGHW(si00, vtmp0, vtmp0) /* si0 = ns0 ns0 ns0 ns0 */
   VMRGLW(si01, vtmp0, vtmp0) /* si1 = ns1 ns1 ns1 ns1 */
   VMRGHW(si02, vtmp2, vtmp2) /* si2 = ns2 ns2 ns2 ns2 */
   VMRGLW(si03, vtmp2, vtmp2) /* si3 = ns3 ns3 ns3 ns3 */

/**
 Load up second scaler:
```

```
**/
   LVX( sval, sptr2, index ) /* read 4 16 bit complex values */
   ADDI(index, index, 16)
   VSUBSHS( neg sval, zero, sval ) /* negate complex scaler values */
   VMRGHW(vtmp0, sval, sval)       /* vtmp0 = s0 s0 s1 s1 */
   VMRGLW(vtmp2, sval, sval)       /* vtmp2 = s2 s2 s3 s3 */
   VMRGHW(sr10, vtmp0, vtmp0) /* sr0 = s0 s0 s0 s0 */
   VMRGLW(sr11, vtmp0, vtmp0) /* sr1 = s1 s1 s1 s1 */
   VMRGHW(sr12, vtmp2, vtmp2) /* sr2 = s2 s2 s2 s2 */
   VMRGLW(sr13, vtmp2, vtmp2) /* sr3 = s3 s3 s3 s3 */

   VPERM( neg sval, sval, neg sval, vc ) /* si -sr */
   VMRGHW(vtmp0, neg sval, neg sval) /* vtmp0 = ns0 ns0 ns1 ns1 */
   VMRGLW(vtmp2, neg sval, neg sval) /* vtmp2 = ns2 ns2 ns3 ns3 */
   VMRGHW(si10, vtmp0, vtmp0) /* si0 = ns0 ns0 ns0 ns0 */
   VMRGLW(si11, vtmp0, vtmp0) /* si1 = ns1 ns1 ns1 ns1 */
   VMRGHW(si12, vtmp2, vtmp2) /* si2 = ns2 ns2 ns2 ns2 */
   VMRGLW(si13, vtmp2, vtmp2) /* si3 = ns3 ns3 ns3 ns3 */
/**
 Assign loop pointers and index registers:
 Loop permute control vector assumes 16 bit input vectors
 C[] -> 16 x N complex elements
 A[] -> 4 x N complex elements
 N -> 4 byte (i.e. interleaved complex) elements
**/
   LVX( vc, tptr, index ) /* interleaves 16 MSBs of real, imaginary */
   LI(aindex, 0)
   LI(cindex, 0)
   ADDI( Cp1, C, 16 )
   ADDI( Cp2, C, 32 )
   ADDI( Cp3, C, 48 )
/**
 Start up loop code:
 Each read on A[] brings in 4 complex input values
**/
   LVX( a00, A1, aindex )
   DECR_C(N)
   LVX( a01, A2, aindex )
   ADDI(aindex, aindex, 16)

   VMSUMSHS( cr00, sr00, a00, zero )
   VMSUMSHS( ci00, si00, a00, zero )
   VMSUMSHS( cr01, sr01, a00, zero )
   VMSUMSHS( ci01, si01, a00, zero )
   VMSUMSHS( cr02, sr02, a00, zero )
   VMSUMSHS( ci02, si02, a00, zero )
   VMSUMSHS( cr03, sr03, a00, zero )
   VMSUMSHS( ci03, si03, a00, zero )
   BEQ( do1 )

   DECR_C(N)
   LVX( a10, A1, aindex )  /* read input for next pass */
   VMSUMSHS( cr00, sr10, a01, cr00 )
   VMSUMSHS( ci00, si10, a01, ci00 )
   LVX( a11, A2, aindex )
   VMSUMSHS( cr01, sr11, a01, cr01 )
   BR( mid_loop0 )
/**
 Top of double loop
**/
LABEL( loop0 )
/* { */
   VMSUMSHS( cr00, sr00, a00, zero )
   VMSUMSHS( ci00, si00, a00, zero )
   VPERM( c2, cr02, ci02, vc )
   STVX( c1, Cp1, cindex )
   VMSUMSHS( cr01, sr01, a00, zero )
```

```
   DECR C(N)
   VMSUMSHS( ci01, si01, a00, zero )
   VMSUMSHS( cr02, sr02, a00, zero )
   VMSUMSHS( ci02, si02, a00, zero )
   VPERM( c3, cr03, ci03, vc )
   STVX( c2, Cp2, cindex )
   VMSUMSHS( cr03, sr03, a00, zero )
   VMSUMSHS( ci03, si03, a00, zero )
   LVX( a10, A1, aindex )  /* read input for next pass */
   VMSUMSHS( cr00, sr10, a01, cr00 )
   VMSUMSHS( ci00, si10, a01, ci00 )
   LVX( a11, A2, aindex )
   STVX( c3, Cp3, cindex )
   VMSUMSHS( cr01, sr11, a01, cr01 )
   ADDI(cindex, cindex, 64)
LABEL( mid loop0 )
   VMSUMSHS( ci01, si11, a01, ci01 )
   VMSUMSHS( cr02, sr12, a01, cr02 )
   VPERM( c0, cr00, ci00, vc ) /* begin permute cycle for this pass */
   STVX( c0, Cp0, cindex ) /* begin write cycle from last pass */
   VMSUMSHS( ci02, si12, a01, ci02 )
   ADDI(aindex, aindex, 16)
   VMSUMSHS( cr03, sr13, a01, cr03 )
   VMSUMSHS( ci03, si13, a01, ci03 )
   VPERM( c1, cr01, ci01, vc )
/* } */
  BNE( loop1 )
/**
 Drop out to flush
**/
   VMSUMSHS( cr00, sr00, a10, zero )
   VMSUMSHS( ci00, si00, a10, zero )
   VPERM( c2, cr02, ci02, vc )
   STVX( c1, Cp1, cindex )
   VMSUMSHS( cr01, sr01, a10, zero )
   VMSUMSHS( ci01, si01, a10, zero )
   VMSUMSHS( cr02, sr02, a10, zero )
   VMSUMSHS( ci02, si02, a10, zero )
   VPERM( c3, cr03, ci03, vc )
   STVX( c2, Cp2, cindex )
   VMSUMSHS( cr03, sr03, a10, zero )
   VMSUMSHS( ci03, si03, a10, zero )
   VMSUMSHS( cr00, sr10, a11, cr00 )
   VMSUMSHS( ci00, si10, a11, ci00 )
   STVX( c3, Cp3, cindex )
   VMSUMSHS( cr01, sr11, a11, cr01 )
   ADDI(cindex, cindex, 64)
   VMSUMSHS( ci01, si11, a11, ci01 )
   VMSUMSHS( cr02, sr12, a11, cr02 )
   VPERM( c0, cr00, ci00, vc ) /* begin permute cycle for this pass */
   STVX( c0, Cp0, cindex ) /* begin write cycle from last pass */
   VMSUMSHS( ci02, si12, a11, ci02 )
   VMSUMSHS( cr03, sr13, a11, cr03 )
   VMSUMSHS( ci03, si13, a11, ci03 )
   VPERM( c1, cr01, ci01, vc )

   VPERM( c2, cr02, ci02, vc )
   STVX( c1, Cp1, cindex )
   VPERM( c3, cr03, ci03, vc )
   STVX( c2, Cp2, cindex )
   STVX( c3, Cp3, cindex )
   BR( ret )
/**
 Top of second loop
**/
LABEL( loop1 )
/* { */
```

gen_x_row.mac 2/23/2001

```
    VMSUMSHS( cr00, sr00, a10, zero )
    VMSUMSHS( ci00, si00, a10, zero )
    VPERM( c2, cr02, ci02, vc )
    STVX( c1, Cp1, cindex )
    VMSUMSHS( cr01, sr01, a10, zero )
    DECR C(N)
    VMSUMSHS( ci01, si01, a10, zero )
    VMSUMSHS( cr02, sr02, a10, zero )
    VMSUMSHS( ci02, si02, a10, zero )
    VPERM( c3, cr03, ci03, vc )
    STVX( c2, Cp2, cindex )
    VMSUMSHS( cr03, sr03, a10, zero )
    VMSUMSHS( ci03, si03, a10, zero )
    LVX( a00, A1, aindex )  /* read input for next pass */
    VMSUMSHS( cr00, sr10, a11, cr00 )
    VMSUMSHS( ci00, si10, a11, ci00 )
    LVX( a01, A2, aindex )
    STVX( c3, Cp3, cindex )
    VMSUMSHS( cr01, sr11, a11, cr01 )
    ADDI(cindex, cindex, 64)
    VMSUMSHS( ci01, si11, a11, ci01 )
    VMSUMSHS( cr02, sr12, a11, cr02 )
    VPERM( c0, cr00, ci00, vc ) /* begin permute cycle for this pass */
    STVX( c0, Cp0, cindex ) /* begin write cycle from last pass */
    VMSUMSHS( ci02, si12, a11, ci02 )
    ADDI(aindex, aindex, 16)
    VMSUMSHS( cr03, sr13, a11, cr03 )
    VMSUMSHS( ci03, si13, a11, ci03 )
    VPERM( c1, cr01, ci01, vc )
/* } */
   BNE( loop0 )
/**
 Flush loop
**/
    VMSUMSHS( cr00, sr00, a00, zero )
    VMSUMSHS( ci00, si00, a00, zero )
    VPERM( c2, cr02, ci02, vc )
    STVX( c1, Cp1, cindex )
    VMSUMSHS( cr01, sr01, a00, zero )
    VMSUMSHS( ci01, si01, a00, zero )
    VMSUMSHS( cr02, sr02, a00, zero )
    VMSUMSHS( ci02, si02, a00, zero )
    VPERM( c3, cr03, ci03, vc )
    STVX( c2, Cp2, cindex )
    VMSUMSHS( cr03, sr03, a00, zero )
    VMSUMSHS( ci03, si03, a00, zero )
    VMSUMSHS( cr00, sr10, a01, cr00 )
    VMSUMSHS( ci00, si10, a01, ci00 )
    STVX( c3, Cp3, cindex )
    VMSUMSHS( cr01, sr11, a01, cr01 )
    ADDI(cindex, cindex, 64)
    VMSUMSHS( ci01, si11, a01, ci01 )
    VMSUMSHS( cr02, sr12, a01, cr02 )
    VPERM( c0, cr00, ci00, vc ) /* begin permute cycle for this pass */
    STVX( c0, Cp0, cindex ) /* begin write cycle from last pass */
    VMSUMSHS( ci02, si12, a01, ci02 )
    VMSUMSHS( cr03, sr13, a01, cr03 )
    VMSUMSHS( ci03, si13, a01, ci03 )
    VPERM( c1, cr01, ci01, vc )

    VPERM( c2, cr02, ci02, vc )
    STVX( c1, Cp1, cindex )
    VPERM( c3, cr03, ci03, vc )
    STVX( c2, Cp2, cindex )
    STVX( c3, Cp3, cindex )
    BR( ret )
```

gen_x_row.mac 2/23/2001

```
LABEL( do1 )
   VMSUMSHS( cr00, sr10, a01, cr00 )
   VMSUMSHS( ci00, si10, a01, ci00 )
   VMSUMSHS( cr01, sr11, a01, cr01 )
   VMSUMSHS( ci01, si11, a01, ci01 )
   VMSUMSHS( cr02, sr12, a01, cr02 )
   VPERM( c0, cr00, ci00, vc ) /* begin permute cycle for this pass */
   STVX( c0, Cp0, cindex ) /* begin write cycle from last pass */
   VMSUMSHS( ci02, si12, a01, ci02 )
   VMSUMSHS( cr03, sr13, a01, cr03 )
   VMSUMSHS( ci03, si13, a01, ci03 )
   VPERM( c1, cr01, ci01, vc )
   VPERM( c2, cr02, ci02, vc )
   STVX( c1, Cp1, cindex )
   VPERM( c3, cr03, ci03, vc )
   STVX( c2, Cp2, cindex )
   STVX( c3, Cp3, cindex )

/**
 Return
**/
LABEL( ret )
  FREE THRU_v31( VRSAVE_COND )
  RETURN
FUNC_EPILOG
```

get_sizes.c 2/23/2001

```
#include "mudlib.h"

/*
 *  Return the offset in units of complex elements into the Corr0 matrix
 *  corresponding to a specified starting physical user and starting virtual
 *  user (within the starting physical user) pair.
 */
int  mudlib get Corr0 offset (
            unsigned char  *ptov_map, /* no more than 256 virts. per phys */
            int  num fingers,         /* typically, 4 */
            int  tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int  start phys user,     /* zero-based index into ptov map */
            int  start_virt_user      /* must be < ptov_map[start_phys_user]
            */
          )
{
  int  num_Corrs, num_virt_users;

  num virt users = mudlib_get_num_virt_users( ptov_map, 0, 0,
  start_phys_user,
                                                start_virt_user ) - 1;
  num_Corrs = (num virt users * tot virt users) -
              ((num_virt_users * (num_virt_users + 1)) / 2);

  return ( num_Corrs * (num_fingers * num_fingers) );
}

/*
 *  Return the size (in bytes) of the portion of the Corr0 matrix
 *  corresponding to a specified starting physical user, virtual
 *  user (within the starting physical user) pair and an ending physical
 *  user, virtual user pair, inclusive. Elements of Corr0 are assumed
 *  to be of type COMPLEX_BF8.
 */
int  mudlib get Corr0 size (
            unsigned char  *ptov_map, /* no more than 256 virts. per phys */
            int  num fingers,         /* typically, 4 */
            int  tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int  start phys user,     /* zero-based index into ptov map */
            int  start_virt_user,     /* must be < ptov_map[start_phys_user]
            */
            int  end phys user,       /* zero-based index into ptov map */
            int  end_virt_user        /* must be < ptov_map[end_phys_user] */
          )
{
  int  start_offset, end_offset;

  start_offset = mudlib_get_Corr0_offset ( ptov map,
                                           num fingers,
                                           tot virt users,
                                           start phys user,
                                           start_virt_user );

  MUDLIB_INCR_VIRT_USER( ptov_map, end_phys_user, end_virt_user )

  end_offset = mudlib_get_Corr0_offset ( ptov map,
                                         num fingers,
                                         tot virt users,
                                         end phys user,
                                         end_virt_user );

  return ( (end_offset - start_offset) * sizeof(COMPLEX_BF8) );
}
```

get_sizes.c 2/23/2001

```
/*
 *  Return the offset in units of complex elements into the Corr1 matrix
 *  corresponding to a specified starting physical user and starting virtual
 *  user (within the starting physical user) pair.
 */
int  mudlib get Corr1 offset (
           unsigned char  *ptov_map, /* no more than 256 virts. per phys */
           int  num fingers,         /* typically, 4 */
           int  tot_virt_users,      /* sum of ptov_map over all phys users
           */
           int  start phys user,     /* zero-based index into ptov map */
           int  start_virt_user      /* must be < ptov_map[start_phys_user]
           */
         )
{
  int  num_Corrs, num_virt_users;

  num virt users = mudlib_get_num_virt_users( ptov_map, 0, 0,
  start_phys_user,
                                              start_virt_user ) - 1;
  num_Corrs = (num_virt_users * tot_virt_users);

  return ( num_Corrs * (num_fingers * num_fingers) );
}

/*
 *  Return the size (in bytes) of the portion of the Corr1 matrix
 *  corresponding to a specified starting physical user, virtual
 *  user (within the starting physical user) pair and an ending physical
 *  user, virtual user pair, inclusive. Elements of Corr1 are assumed
 *  to be of type COMPLEX_BF8.
 */
int  mudlib get Corr1 size (
           unsigned char  *ptov_map, /* no more than 256 virts. per phys */
           int  num fingers,         /* typically, 4 */
           int  tot_virt_users,      /* sum of ptov_map over all phys users
           */
           int  start phys user,     /* zero-based index into ptov map */
           int  start_virt_user,     /* must be < ptov_map[start_phys_user]
           */
           int  end phys user,       /* zero-based index into ptov map */
           int  end_virt_user        /* must be < ptov_map[end_phys_user] */
         )
{
  int  start_offset, end_offset;

  start_offset = mudlib_get_Corr1_offset ( ptov map,
                                           num fingers,
                                           tot virt users,
                                           start phys user,
                                           start_virt_user );

  MUDLIB_INCR_VIRT_USER( ptov_map, end_phys_user, end_virt_user )

  end_offset = mudlib_get_Corr1_offset ( ptov map,
                                         num fingers,
                                         tot virt users,
                                         end phys user,
                                         end_virt_user );

  return ( (end_offset - start_offset) * sizeof(COMPLEX_BF8) );
}

/*
 *  Return the offset into the R0 matrix corresponding to a specified
 *  starting physical user and starting virtual user (within the
 *  starting physical user) pair.
```

```
 */
int  mudlib get R0 offset (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        )
{
  int  i, num_virt_users, offset, tcols;

  tcols = (tot virt users + R MATRIX ALIGN MASK) & ~R MATRIX ALIGN_MASK;
  num virt users = mudlib_get_num_virt_users( ptov_map, 0, 0,
  start_phys_user,
                                                start_virt_user ) - 1;
  offset = 0;
  for ( i = 0; i < num_virt_users; i++ )
    offset += (tcols - (i & ~R_MATRIX_ALIGN_MASK));
  return offset;
}

/*
 *  Return the size (in bytes) of the portion of the R0 matrix
 *  corresponding to a specified starting physical user, virtual
 *  user (within the starting physical user) pair and an ending physical
 *  user, virtual user pair, inclusive. Elements of R0 are assumed
 *  to be of type BF8.
 */
int  mudlib get R0 size (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user,     /* must be < ptov_map[start_phys_user]
          */
          int  end phys user,       /* zero-based index into ptov map */
          int  end_virt_user        /* must be < ptov_map[end_phys_user] */
        )
{
  int  start_offset, end_offset;

  start_offset = mudlib_get_R0_offset ( ptov map,
                                        tot virt users,
                                        start phys user,
                                        start_virt_user );

  MUDLIB_INCR_VIRT_USER( ptov_map, end_phys_user, end_virt_user )

  end_offset = mudlib_get_R0_offset ( ptov map,
                                      tot virt users,
                                      end phys user,
                                      end_virt_user );

  return ( (end_offset - start_offset) * sizeof(BF8) );
}

/*
 *  Return the offset into the R1 matrix corresponding to a specified
 *  starting physical user and starting virtual user (within the
 *  starting physical user) pair.
 */
int  mudlib get R1 offset (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start_phys_user,     /* zero-based index into ptov_map */
```

```
            int   start_virt_user      /* must be < ptov_map[start_phys_user]
            */
          )
{
  int  num_virt_users, tcols;

  tcols = (tot virt users + R MATRIX ALIGN MASK) & ~R MATRIX ALIGN_MASK;
  num virt users = mudlib_get_num_virt_users( ptov_map, 0, 0,
  start_phys_user,
                                                    start_virt_user ) - 1;
  return ( num_virt_users * tcols );
}

/*
 *  Return the size (in bytes) of the portion of the R1 matrix
 *  corresponding to a specified starting physical user, virtual
 *  user (within the starting physical user) pair and an ending physical
 *  user, virtual user pair, inclusive. Elements of R1 are assumed
 *  to be of type BF8.
 */
int  mudlib get R1 size (
            unsigned char  *ptov_map, /* no more than 256 virts. per phys */
            int  tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int  start phys user,     /* zero-based index into ptov map */
            int  start_virt_user,     /* must be < ptov_map[start_phys_user]
            */
            int  end phys user,       /* zero-based index into ptov map */
            int  end_virt_user        /* must be < ptov_map[end_phys_user] */
          )
{
  int  start_offset, end_offset;

  start_offset = mudlib_get_R1_offset ( ptov map,
                                        tot virt users,
                                        start phys user,
                                        start_virt_user );

  MUDLIB_INCR_VIRT_USER( ptov_map, end_phys_user, end_virt_user )

  end_offset = mudlib_get_R1_offset ( ptov map,
                                      tot virt users,
                                      end phys user,
                                      end_virt_user );

  return ( (end_offset - start_offset) * sizeof(BF8) );
}

/*
 *  Return the number of virtual users
 *  corresponding to a specified starting physical user, virtual
 *  user (within the starting physical user) pair and an ending physical
 *  user, virtual user pair, inclusive.
 */
int  mudlib get num_virt_users (
            unsigned char  *ptov map, /* no more than 256 virts. per phys */
            int  start phys user,     /* zero-based index into ptov map */
            int  start_virt_user,     /* must be < ptov_map[start_phys_user]
            */
            int  end phys user,       /* zero-based index into ptov map */
            int  end_virt_user        /* must be < ptov_map[end_phys_user] */
          )
{
  int  i, num_virt_users;

  if ( start_phys user == end phys user )
    return ( end_virt_user - start_virt_user + 1 );
```

get_sizes.c 2/23/2001

```
  else {
    num_virt users = ptov map[start phys_user] - start virt_user;
    for ( i = (start_phys user + 1); i < end_phys_user; i++ )
      num virt users += ptov map[i];
    num virt_users += (end virt_user + 1);
    return ( num_virt_users );
  }
}

/*
 *  For a specified starting physical user, virtual user
 *  (within the starting physical user) pair and a specified
 *  number of virtual users inclusive of the starting pair,
 *  return (in separate arguments), the corresponding ending
 *  physical user, virtual user pair (inclusive).
 */
void  mudlib get end user_pair (
           unsigned char  *ptov map, /* no more than 256 virts. per phys */
           int  start phys user,     /* zero-based index into ptov map */
           int  start_virt_user,     /* must be < ptov_map[start_phys_user]
           */
           int  num virt users,      /* number from start (must be > 0) */
           int  *end phys user,      /* zero-based index into ptov map */
           int  *end_virt_user       /* will be < ptov_map[*end_phys_user] */
         )
{
  int  i, j;

  for ( i = start phys user; ; i++ ) {
    for ( j = start virt user; j < ptov map[i]; j++ )
      if ( --num virt users == 0 ) break;
    if ( num virt users == 0 ) break;
    start_virt_user = 0;
  }

  *end phys user = i;
  *end_virt_user = j;
}
```

get_sizes_v.c 2/23/2001

```
#include "mudlib.h"

/*************************************************************************
 * Virtual users version
 ************************************************************************/

int  mudlib_get_Corr0_offset_v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num_fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start_phys_user,     /* zero-based index into ptov_map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        )
{
  int  i, num_fingers_squared, remaining_size, skipped_virt_users,
  total_size;

  num_fingers_squared = num_fingers * num_fingers;
  skipped_virt_users = 0;

  for ( i = 0; i < start_phys_user; i++ )
    skipped_virt_users += (int)ptov_map[i];

  skipped_virt_users += start_virt_user;

  // Always even
  total_size = tot_virt_users * ( tot_virt_users - 1 );
  remaining_size = ( tot_virt_users - skipped_virt_users )
                 * ( tot_virt_users - skipped_virt_users - 1 );

  // zero based units of complex elements
  return ( num_fingers_squared * ( ( total_size - remaining_size ) >> 1 ) );
}


int  mudlib_get_Corr1_offset_v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num_fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start_phys_user,     /* zero-based index into ptov_map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        )
{
  int  i, num_fingers_squared, skipped_virt_users;

  num_fingers_squared = num_fingers * num_fingers;
  skipped_virt_users = 0;

  for ( i = 0; i < start_phys_user; i++ )
    skipped_virt_users += (int)ptov_map[i];

  skipped_virt_users += start_virt_user;

  return ( num_fingers_squared * ( skipped_virt_users * tot_virt_users ) );
}


int  mudlib_get_R0_offset_v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
```

```
            int   start phys user,      /* zero-based index into ptov map */
            int   start_virt_user       /* must be < ptov_map[start_phys_user]
            */
          )
{
  int  i, iv;
  int  R0_skipped_virt_users, R0_tcols, tcols, size;

  tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;

  R0_skipped_virt_users = 0;

  size = 0;

  for ( i = 0; i < start phys user; i++ ) {
    for ( iv = 0; iv < (int)ptov_map[i]; iv++ ) {

      R0_tcols = tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);

      size += R0 tcols;
      ++R0_skipped_virt_users;
    }
  }

  /* Handle last physical user, potentially split on virt users */

  for ( iv = 0; iv < (int) start_virt_user; iv++ ) {

    R0_tcols = tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);

    size += R0 tcols;
    ++R0_skipped_virt_users;
  }

  return size;
}


int  mudlib get R0 size v (
            unsigned char  *ptov_map,  /* no more than 256 virts. per phys */
            int   tot_virt_users,       /* sum of ptov_map over all phys users
            */
            int   start phys user,      /* zero-based index into ptov map */
            int   start_virt_user,      /* must be < ptov_map[start_phys_user]
            */
            int   end phys user,        /* zero-based index into ptov map */
            int   end_virt_user         /* must be < ptov_map[end_phys_user] */
          )
{
  int  i, iv;
  int  R0_skipped_virt_users, R0_tcols, tcols, size;

  tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;

  R0 skipped virt users = 0;
  for ( i = 0; i < start phys user; i++ )
    R0_skipped_virt_users += (int)ptov_map[i];

  R0_skipped_virt_users += start_virt_user;

  //  printf("skipped:  %d\n", R0_skipped_virt_users);

  size = 0;

  if ( start_phys_user == end_phys_user )
  {
    //    printf("start == end phys\n");
```

```
    // <= for Inclusive
    for ( iv = start_virt_user; iv <= (int) end_virt_user; iv++ ) {

      R0_tcols = tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);

      size += R0 tcols;
      //  printf("size:  %d, R0tc:  %d\n", size, R0_tcols);
      ++R0_skipped_virt_users;
    }
  }
  else
  {
    for ( i = start_phys_user; i < end phys user; i++ ) {
      for ( iv = 0; iv < (int)ptov_map[i]; iv++ ) {

        R0_tcols = tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);

        size += R0 tcols;
        //  printf("size:  %d, R0tc:  %d\n", size, R0_tcols);
        ++R0_skipped_virt_users;
      }
    }

    /* Handle last physical user, potentially split on virt users */
    //     printf("last phys user \n");
    // <= for Inclusive
    for ( iv = start_virt_user; iv <= (int) end_virt_user; iv++ ) {

      R0_tcols = tcols - (R0_skipped_virt_users & ~R_MATRIX_ALIGN_MASK);

      size += R0 tcols;
      //       printf("size:  %d, R0tc:  %d\n", size, R0_tcols);
      ++R0_skipped_virt_users;
    }
  }

  return size;
}


int  mudlib get R1 offset_v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        )
{
  int  i, tcols, virt_users;

  tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;

  virt_users = 0;

  // Main loop
  for ( i = 0;  i < start phys user; i++ ) {
    virt_users += (int)ptov_map[i];
  }

  // Trailing virtual users
  virt_users += start_virt_user;

  return ( virt_users * tcols );
}
```

get_sizes_v.c 2/23/2001

```
int  mudlib get R1 size v (
           unsigned char  *ptov_map, /* no more than 256 virts. per phys */
           int  tot_virt_users,      /* sum of ptov_map over all phys users
           */
           int  start phys user,     /* zero-based index into ptov map */
           int  start_virt_user,     /* must be < ptov_map[start_phys_user]
           */
           int  end phys user,       /* zero-based index into ptov map */
           int  end_virt_user        /* must be < ptov_map[end_phys_user] */
         )
{
  int  i, tcols, virt_users;

  tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;

  virt_users = 0;

  if ( start_phys_user == end_phys_user )
  {
    virt_users = end_virt_user - start_virt_user + 1;
  }
  else if (start_phys_user < end_phys_user)
  {
    // Leading virtual users
    virt_users = (int) ptov_map[start_phys_user] - start_virt_user;

    // Main loop
    for ( i = (start phys user + 1); i < end_phys_user; i++ )
      virt_users += (int)ptov_map[i];

    // Trailing virtual users
    virt_users += (end_virt_user + 1);
  }

  return ( virt_users * tcols );
}
```

2/23/2001

```
#define IO 1
#define TIME 0

//
//  Asynchronous MPIC
//
#if TIME
#include <tmr.h>
#endif

#include "mudlib.h"

void sve3_8bit( BF8 *A, BF8 *B, BF8 *C, BF32 *sum, int n );

void dotpr3_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols );

void dotpr6_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols );

void dotpr9_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols );

#if TIME
static int time_count = 0;
static int z;
static float  time;
static TMR ts  time0, time1;
static TMR_timespec  elapsed;
#endif

/*
 *  void async multirate mpic ( BF8 *Bt hat, BF8 *R0 hat,
 *                              BF8 *R1 hat, BF8 *R1m hat,
 *                              BF32 *Y, BF32 Ythresh,
 *                              int N_users, int N_bits, int N_stages )
 *
 *  N users must be > 0 and divisible by 4
 *  N_bits must be >= 5
 */

void  mudlib_mpic ( BF8 *Bt hat,
                    BF8 *R0 hat,
                    BF8 *R1 hat,
                    BF8 *R1m_hat,
                    BF32 *Y,
                    BF32 Ythresh,
                    int N users,
                    int N bits,
                    int N_stages )

{
  BF8  *Bt hatp;
  BF8  *R0 hatp, *R1_hatp, *R1m_hatp;
  BF32  *Yp;
  BF32  R bias, sums[3];
  int  hat_tc, i, m, N_users_pad, stage;

  hat tc = (N_users + R MATRIX ALIGN MASK) & ~R MATRIX ALIGN MASK;
  N_users_pad = (N_users + ALTIVEC_ALIGN_MASK) & ~ALTIVEC_ALIGN_MASK;

#if 0
  if ( ( (long)Bt hat | (long)R0 upper bf | (long)R0 lower bf |
         (long)R1 trans bf | (long)R1m bf) & ALTIVEC ALIGN_MASK ) {
    printf ( "***** inputs are NON-ALIGNED *****\n" );
    exit( -1 );
```

```
  }
#endif

  //
  //  Subtract interference in N_stages
  //
  for ( stage = 0; stage < N_stages; stage++ ) {

    R0 hatp = R0 hat;
    R1 hatp = R1 hat;
    R1m hatp = R1m_hat;
    Yp = Y;

    for ( i = 0; i < N_users; i++ ) {

      sve3_8bit( R0_hatp, R1_hatp, R1m_hatp, &R_bias, N_users_pad );

#if 0
      R0_hatp[i] = BF8_ZERO;           /* zero diagonal element */
#endif

      Bt hatp = Bt_hat + hat_tc;       /* points to leading row */
      m = 2;

      while ( m < (N bits-4) ) {
        if ( BFABS( Yp[m] ) < Ythresh ) {
          if ( BFABS( Yp[m+1] ) < Ythresh ) {
            if ( BFABS( Yp[m+2] ) < Ythresh ) {
              dotpr9_8bit( Bt hatp, R1 hatp, R0 hatp, R1m_hatp,
                           sums, N_users_pad, hat_tc );
              sums[0] -= R bias;
              sums[1] -= ((BF32)Bt hatp[hat tc + i] * (BF32)R1_hatp[i]);
              if ( (Yp[m] - sums[0]) > BF32 ZERO )
                Bt_hatp[hat_tc + i] = 1 + BIAS_8BIT;
              else
                Bt hatp[hat tc + i] = -1 + BIAS 8BIT;
              sums[1] += ((BF32)Bt_hatp[hat_tc + i] * (BF32)R1_hatp[i]);

              sums[1] -= R bias;
              sums[2] -= ((BF32)Bt hatp[2*hat tc + i] * (BF32)R1_hatp[i]);
              if ( (Yp[m+1] - sums[1]) > BF32 ZERO )
                Bt_hatp[2*hat_tc + i] = 1 + BIAS_8BIT;
              else
                Bt hatp[2*hat tc + i] = -1 + BIAS 8BIT;
              sums[2] += ((BF32)Bt_hatp[2*hat_tc + i] * (BF32)R1_hatp[i]);

              sums[2] -= R bias;
              if ( (Yp[m+2] - sums[2]) > BF32 ZERO )
                Bt_hatp[3*hat_tc + i] = 1 + BIAS_8BIT;
              else
                Bt_hatp[3*hat_tc + i] = -1 + BIAS_8BIT;
            }
            else {                       /* skip third sum */
              dotpr6_8bit( Bt hatp, R1 hatp, R0 hatp, R1m_hatp,
                           sums, N_users_pad, hat_tc );
              sums[0] -= R bias;
              sums[1] -= ((BF32)Bt hatp[hat tc + i] * (BF32)R1_hatp[i]);
              if ( (Yp[m] - sums[0]) > BF32 ZERO )
                Bt_hatp[hat_tc + i] = 1 + BIAS_8BIT;
              else
                Bt hatp[hat tc + i] = -1 + BIAS 8BIT;
              sums[1] += ((BF32)Bt_hatp[hat_tc + i] * (BF32)R1_hatp[i]);

              sums[1] -= R bias;
              if ( (Yp[m+1] - sums[1]) > BF32 ZERO )
                Bt_hatp[2*hat_tc + i] = 1 + BIAS_8BIT;
              else
```

```
                Bt_hatp[2*hat_tc + i] = -1 + BIAS_8BIT;
            }
#if IO
            Bt_hatp += hat_tc;              /* bump leading row pointer */
#endif
            ++m;                            /* bump row */
          }
          else {                            /* skip second sum */
            dotpr3_8bit( Bt hatp, R1 hatp, R0 hatp, R1m_hatp,
                         sums, N_users_pad, hat_tc );
            sums[0] -= R bias;
            if ( (Yp[m] - sums[0]) > BF32 ZERO )
              Bt_hatp[hat_tc + i] = 1 + BIAS_8BIT;
            else
              Bt_hatp[hat_tc + i] = -1 + BIAS_8BIT;
          }
#if IO
          Bt_hatp += hat_tc;                /* bump leading row pointer */
#endif
          ++m;                              /* bump row */
        }

#if IO
        Bt_hatp += hat_tc;                  /* bump leading row pointer */
#endif
        ++m;                                /* bump row */
      }

      /*
       *  do last 0, 1 or 2 dot product calculations
       */
      while ( m < (N bits-2) ) {
        if ( BFABS( Yp[m] ) < Ythresh ) {
          dotpr3_8bit( Bt hatp, R1 hatp, R0 hatp, R1m_hatp,
                       sums, N_users_pad, hat_tc );
          sums[0] -= R bias;
          if ( (Yp[m] - sums[0]) > BF32 ZERO )
            Bt_hatp[hat_tc + i] = 1 + BIAS_8BIT;
          else
            Bt_hatp[hat_tc + i] = -1 + BIAS_8BIT;
        }

#if IO
        Bt_hatp += hat_tc;                  /* bump leading row pointer */
#endif
        ++m;
      }

#if IO
      R0 hatp += hat tc;                /* bump pointer */
      R1 hatp += hat tc;                /* bump pointer */
      R1m hatp += hat_tc;               /* bump pointer */
      Yp += N_bits;                     /* bump pointer */
#endif
    }                                   /* end of loop over N users */
  }                                     /* end of loop over N_stages */
}

#if defined( COMPILE_C )

void dotpr3_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols )
{
  int  j;

  sums[0] = BF32_ZERO;
  for ( j = 0; j < N; j++ ) {
```

```
    sums[0] += (BF32)A[j] * (BF32)B0[j];
    sums[0] += (BF32)A[tcols+j] * (BF32)B1[j];
    sums[0] += (BF32)A[(tcols<<1)+j] * (BF32)B2[j];
  }
}

void dotpr6_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols )
{
  int  i, j;

  for ( i = 0; i < 2; i++ ) {
    sums[i] = BF32_ZERO;
    for ( j = 0; j < N; j++ ) {
      sums[i] += (BF32)A[i*tcols + j] * (BF32)B0[j];
      sums[i] += (BF32)A[(i+1)*tcols + j] * (BF32)B1[j];
      sums[i] += (BF32)A[(i+2)*tcols + j] * (BF32)B2[j];
    }
  }
}

void dotpr9_8bit( BF8 *A, BF8 *B0, BF8 *B1, BF8 *B2,
                  BF32 *sums, int N, int tcols )
{
  int  i, j;

  for ( i = 0; i < 3; i++ ) {
    sums[i] = BF32_ZERO;
    for ( j = 0; j < N; j++ ) {
      sums[i] += (BF32)A[i*tcols + j] * (BF32)B0[j];
      sums[i] += (BF32)A[(i+1)*tcols + j] * (BF32)B1[j];
      sums[i] += (BF32)A[(i+2)*tcols + j] * (BF32)B2[j];
    }
  }
}

void sve3_8bit( BF8 *A, BF8 *B, BF8 *C, BF32 *sum, int n )
{
  int  i;
  BF32  wsum;

  wsum = 0;
  for ( i = 0; i < n; i++ ) {
    wsum += (BF32)A[i];
    wsum += (BF32)B[i];
    wsum += (BF32)C[i];
  }
  *sum = wsum;
}

#endif
```

gen_r_matrices.mac 2/23/2001

```
/*-----------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version      ---|
|------------------------------------------------------------------|
|                                                                  |
|  File Name:    GEN R_MATRICES.MAC                                |
|  Description:  Float and scale R matrix values, convert to byte.|
|                                                                  |
|  Entry/params: GEN_R_MATRICES( Rsump, Bf scalep, Inv scalep,    |
|                                Scalep, No scale row bfp,         |
|                                Scale_row_bfp, Num_virt_users )   |
|                                                                  |
|  Formula:                                                        |
|                                                                  |
|   bf scale = *bf scalep;                                         |
|   inv_scale = *inv_scalep;                                       |
|                                                                  |
|   for ( i = 0; i < num_virt_users; i++ ) {                       |
|     scale = scalep[i];                                           |
|     fsum = (float)(R sums[i]);                                   |
|     fsum *= bf_scale;                                            |
|                                                                  |
|     fsum scale = fsum * inv_scale;                               |
|     fsum_scale *= scale;                                         |
|                                                                  |
|     SATURATE( fsum_scale )                                       |
|     SATURATE( fsum )                                             |
|                                                                  |
|     no scale row bfp[i] = BF8 FIX( fsum );                       |
|     scale_row_bfp[i] = BF8_FIX( fsum_scale );                    |
|   }                                                              |
|                                                                  |
|            Mercury Computer Systems, Inc.                        |
|            Copyright (c) 2000  All rights reserved               |
|                                                                  |
|   Revision      Date     Engineer  Reason                        |
|   --------      ----     --------  --------                      |
|     0.0        000910       fpl    Created                       |
|     0.1        000914       fpl    Removed VMAXFP and added      |
|                                    windin code                   |
|     0.3        000920       fpl    Removed all windin and windout|
+-----------------------------------------------------------------*/

#include "salppc.inc"

#define DO_IO  1

#if DO IO
#define  SCALE_BUMP_16  16
#else
#define  SCALE_BUMP_16  0
#endif

#define STORE_SCALE( vS, rA, rB )   STVX( vS, rA, rB )

#define ZERO_COND  6

RODATA_SECTION( 6 )

START_L_ARRAY( local_table )

/**
 First stage for byte pack
**/
L_PERMUTE_MASK( 0x0004080c, 0x1014181c, 0x0004080c, 0x1014181c )

/**
```

```
 Second stage for byte pack
**/
L_PERMUTE_MASK( 0x00010203, 0x04050607, 0x10111213, 0x14151617 )

END_ARRAY

/**
 Input parameters
**/
#define Rsump              r3
#define Bf scalep          r4
#define Inv scalep         r5
#define Scalep             r6
#define No scale row bfp   r7
#define Scale row bfp      r8
#define Num_virt_users     r9

/**
 Local GPRs
**/
#define indx1              r10
#define indx2              r11
#define indx3              r12

#define low4                r0

#define tptr              indx2
#define low4x4            low4

/**
 G4 registers
**/
#define zero             v0
#define inv scale        v1
#define bf_scale         v2

#define byte pack        v3
#define byte_merge       v4

#define scale0           v5
#define scale1           v6
#define vtmp           scale1
#define scale2           v7
#define vtmp2          scale2
#define scale3           v8

#define fsum0            v9
#define fsum1            v10
#define fsum2            v11
#define fsum3            v12

#define fsum scale0      v13
#define fsum scale1      v14
#define fsum scale2      v15
#define fsum_scale3      v16

#define bsum0            v17
#define bsum1            v18
#define bsum2            v19
#define bsum3            v20

#define bsum scale0      v21
#define bsum scale1      v22
#define bsum scale2      v23
#define bsum_scale3      v24

#define bvector          v25
```

gen_r_matrices.mac 2/23/2001

```
#define bscale_vector  v26

#define rsum0          v27
#define rsum1          v28
#define rsum2          v29
#define rsum3          v30
#define seven          v31

/**
 Begin code text
**/
FUNC_PROLOG

ENTRY_7( gen R matrices, Rsump, Bf scalep, Inv scalep, Scalep, \
         No_scale_row_bfp, Scale_row_bfp, Num_virt_users )

  CMPWI( Num_virt_users, 0 )
  BGT( start )
  RETURN

LABEL( start )
  USE_THRU_v31( VRSAVE_COND )
/**
 Load up permute vectors and loop scalers
**/
   LA( tptr, local_table, 0 )
   LI( indx1, 16 )
   LVX( byte_pack, 0, tptr )
   VSPLTISB( seven, 7 )
   LVX( byte merge, tptr, indx1 )
   SCALAR SPLAT( bf scale, vtmp, Bf scalep )
   SCALAR_SPLAT( inv_scale, vtmp, Inv_scalep )

/**
 Back up to nearest 16-byte boundary. It's okay to write before and after to
 nearest 16-byte boundary in both directions.
**/
   RLWINM( low4, No scale_row_bfp, 0, 28, 31 )     /* lower 4 bits */
   VXOR( zero, zero, zero )
   ADD( Num virt users, Num virt users, low4 )
   SUB( No scale row bfp, No scale row bfp, low4 )
   SUB( Scale row bfp, Scale_row_bfp, low4 )
   SLWI( low4x4, low4, 2 )
   LI( indx2, 32 )
   SUB( Rsump, Rsump, low4x4 )

/**
 Start up loop
**/
   LVX( rsum0, 0, Rsump )
   LI( indx3, 48 )
   LVX( rsum1, Rsump, indx1 )
   SUB( Scalep, Scalep, low4x4 )
   LVX( rsum2, Rsump, indx2 )
   VCFSX( fsum0, rsum0, 0 )
   LVX( rsum3, Rsump, indx3 )
   VCFSX( fsum1, rsum1, 0 )
   LVX( scale0, 0, Scalep )
   VCFSX( fsum2, rsum2, 0 )
   LVX( scale1, Scalep, indx1 )
   VCFSX( fsum3, rsum3, 0 )
   LVX( scale2, Scalep, indx2 )
   VMADDFP( fsum0, fsum0, bf scale, zero )
   LVX( scale3, Scalep, indx3 )
   VMADDFP( fsum1, fsum1, bf scale, zero )
   ADDIC C( Num virt users, Num virt users, -16 )
   VMADDFP( fsum2, fsum2, bf_scale, zero )
```

gen_r_matrices.mac 2/23/2001

```
    VMADDFP( fsum3, fsum3, bf scale, zero )
    VMADDFP( fsum scale0, fsum0, inv scale, zero )
    VMADDFP( fsum scale1, fsum1, inv scale, zero )
    VMADDFP( fsum scale2, fsum2, inv_scale, zero )
    ADDI( Rsump, Rsump, 64 )
    VMADDFP( fsum_scale3, fsum3, inv_scale, zero )
    ADDI( Scalep, Scalep, 64 )
    VMADDFP( fsum scale0, fsum scale0, scale0, zero )
    VMADDFP( fsum scale1, fsum scale1, scale1, zero )
    VMADDFP( fsum scale2, fsum scale2, scale2, zero )
    VMADDFP( fsum scale3, fsum_scale3, scale3, zero )
    BLE( sixteen_sums )

    LVX( rsum0, 0, Rsump )
    LVX( rsum1, Rsump, indx1 )
    VCTSXS( bsum0, fsum0, 24 )
    LVX( rsum2, Rsump, indx2 )
    VCTSXS( bsum1, fsum1, 24 )
    VCTSXS( bsum2, fsum2, 24 )
    LVX( rsum3, Rsump, indx3 )
    ADDI( Rsump, Rsump, 64 )
    VCTSXS( bsum3, fsum3, 24 )
    LVX( scale0, 0, Scalep )
    VCTSXS( bsum scale0, fsum scale0, 24 )
    VCTSXS( bsum_scale1, fsum scale1, 24 )
    LVX( scale1, Scalep, indx1 )
    VCTSXS( bsum_scale2, fsum scale2, 24 )
    LVX( scale2, Scalep, indx2 )
    ADDI( No scale row bfp, No scale row_bfp, -SCALE_BUMP_16 )
    VCTSXS( bsum scale3, fsum scale3, 24 )
    ADDI( Scale_row_bfp, Scale_row_bfp, -SCALE_BUMP_16 )

    BR( mloop )
/**
 Top of loop outputs 32 bytes per trip
**/
LABEL( loop )
/* { */
     STORE SCALE( bvector, 0, No scale_row bfp )
     VCTSXS( bsum_scale3, fsum scale3, 24 )
     STORE_SCALE( bscale_vector, 0, Scale_row_bfp )

LABEL( mloop )
     LVX( scale3, Scalep, indx3 )
     VCFSX( fsum0, rsum0, 0 )
     VPERM( bsum0, bsum0, bsum1, byte_pack )
     VCFSX( fsum1, rsum1, 0 )
     VCFSX( fsum2, rsum2, 0 )
     ADDI( No scale row bfp, No_scale_row_bfp, SCALE_BUMP_16 )
     VCFSX( fsum3, rsum3, 0 )
     ADDI( Scale row_bfp, Scale row bfp, SCALE_BUMP_16 )
     VMADDFP( fsum0, fsum0, bf scale, zero )
     VPERM( bsum2, bsum2, bsum3, byte_pack )
     VMADDFP( fsum1, fsum1, bf scale, zero )
     VMADDFP( fsum2, fsum2, bf_scale, zero )

     VMADDFP( fsum3, fsum3, bf scale, zero )
     VMADDFP( fsum scale0, fsum0, inv scale, zero )
     VPERM( bvector, bsum0, bsum2, byte merge )
     VMADDFP( fsum scale1, fsum1, inv scale, zero )
     ADDIC C( Num virt users, Num_virt users, -16 )
     VMADDFP( fsum scale2, fsum2, inv scale, zero )
     VMADDFP( fsum_scale3, fsum3, inv_scale, zero )
     ADDI( Scalep, Scalep, 64 )
     VMADDFP( fsum scale0, fsum scale0, scale0, zero )
     VPERM( bsum scale0, bsum scale0, bsum scale1, byte_pack )
     VMADDFP( fsum_scale1, fsum_scale1, scale1, zero )
```

```
    VMADDFP( fsum_scale2, fsum_scale2, scale2, zero )
    VMADDFP( fsum_scale3, fsum_scale3, scale3, zero )
    VPERM( bsum_scale2, bsum_scale2, bsum_scale3, byte_pack )
      VSRB( vtmp, bvector, seven )
    VPERM( bscale_vector, bsum_scale0, bsum_scale2, byte_merge )
      VSRB( vtmp2, bscale_vector, seven )
    BLE( loop_flush )

    LVX( rsum0, 0, Rsump )
      VADDSBS( bvector, bvector, vtmp )
    LVX( rsum1, Rsump, indx1 )
      VADDSBS( bscale_vector, bscale_vector, vtmp2 )
    LVX( rsum2, Rsump, indx2 )
    VCTSXS( bsum0, fsum0, 24 )
    LVX( rsum3, Rsump, indx3 )
    VCTSXS( bsum1, fsum1, 24 )
    ADDI( Rsump, Rsump, 64 )
    VCTSXS( bsum2, fsum2, 24 )
    LVX( scale0, 0, Scalep )
    VCTSXS( bsum3, fsum3, 24 )
    LVX( scale1, Scalep, indx1 )
    VCTSXS( bsum_scale0, fsum_scale0, 24 )
    VCTSXS( bsum_scale1, fsum_scale1, 24 )
    LVX( scale2, Scalep, indx2 )
    VCTSXS( bsum_scale2, fsum_scale2, 24 )
/* } */
  BR( loop )

/**
 Flush loop
**/
LABEL( loop_flush )
     VADDSBS( bvector, bvector, vtmp )
   STORE_SCALE( bvector, 0, No_scale_row_bfp )
     VADDSBS( bscale_vector, bscale_vector, vtmp2 )
   STORE_SCALE( bscale_vector, 0, Scale_row_bfp )
   ADDI( No_scale_row_bfp, No_scale_row_bfp, SCALE_BUMP_16 )
   ADDI( Scale_row_bfp, Scale_row_bfp, SCALE_BUMP_16 )

LABEL( sixteen_sums )

   VCTSXS( bsum0, fsum0, 24 )
   VCTSXS( bsum1, fsum1, 24 )
   VCTSXS( bsum2, fsum2, 24 )
   VCTSXS( bsum3, fsum3, 24 )
   VCTSXS( bsum_scale0, fsum_scale0, 24 )
   VPERM( bsum0, bsum0, bsum1, byte_pack )
   VCTSXS( bsum_scale1, fsum_scale1, 24 )
   VPERM( bsum2, bsum2, bsum3, byte_pack )
   VCTSXS( bsum_scale2, fsum_scale2, 24 )
   VPERM( bvector, bsum0, bsum2, byte_merge )
   VCTSXS( bsum_scale3, fsum_scale3, 24 )

   VPERM( bsum_scale0, bsum_scale0, bsum_scale1, byte_pack )
   VPERM( bsum_scale2, bsum_scale2, bsum_scale3, byte_pack )
     VSRB( vtmp, bvector, seven )
   VPERM( bscale_vector, bsum_scale0, bsum_scale2, byte_merge )
     VADDSBS( bvector, bvector, vtmp )
     VSRB( vtmp, bscale_vector, seven )
   STORE_SCALE( bvector, 0, No_scale_row_bfp )
     VADDSBS( bscale_vector, bscale_vector, vtmp )
   STORE_SCALE( bscale_vector, 0, Scale_row_bfp )
/**
 Return
**/
LABEL( ret )
  FREE_THRU_v31( VRSAVE_COND )
```

```
  RETURN
FUNC_EPILOG
```

m_voter.vhd 3/9/2001

```
--**************************************************************
--**************************************************************
--**
--**  Majority Voter Control Logic
--**
--**  Description: This Module serves as a generic majority voter
--**
--**
--**  Author      : Steven Imperiali/Mirza Cifric
--**  Date        : 5-15-2000
--**
--**
--**************************************************************
LIBRARY IEEE;
USE IEEE.STD LOGIC 1164.ALL;
use ieee.std logic arith.all;
use ieee.std logic unsigned.all;
USE STD.TEXTIO.ALL;


ENTITY m_voter IS
  PORT(
        clk 66 pal6      :IN     std logic;
        reset 0          :IN     std logic;
        request0 0       :IN     std logic;
        request1 0       :IN     std logic;
        request2 0       :IN     std logic;
        request3 0       :IN     std logic;
        request4 0       :IN     std logic;
        healthy0 1       :IN     std logic;
        healthy1 1       :IN     std logic;
        healthy2 1       :IN     std logic;
        healthy3 1       :IN     std logic;
        healthy4 1       :IN     std logic;
        voteout_0        :OUT    std_logic);

END m_voter;

ARCHITECTURE voter OF m voter IS
signal pro: STD_LOGIC VECTOR(3 downto 0);
signal against: STD_LOGIC_VECTOR(3 downto 0);

signal result: STD_LOGIC;

BEGIN



check result:process(request0_0,request1_0,request2_0,request3_0,request4_0,h
ealthy0 1,
healthy1_1,healthy2 1,healthy3 1,healthy4 1)
variable pro: STD_LOGIC VECTOR(3 downto 0);
variable against: STD LOGIC VECTOR(3 downto 0);
variable solution: STD_LOGIC;
begin
    pro:= "0000";                         -- set number of pro voters
    against:="0000";
-- set number of against voters-- Get the number of pros
        if (healthy0_1 = '1' and request0_0='1') then
        pro := pro + "0001";
            end if;
    if (healthy1 1='1' and request1_0='1') then
        pro := pro + "0001";
            end if;
    if (healthy2_1='1' and request2_0='1') then
```

```
        pro := pro + "0001";
            end if;
    if (healthy3 1='1' and request3_0='1') then
        pro := pro + "0001";
            end if;
    if (healthy4 1='1' and request4_0='1') then
        pro := pro + "0001";
            end if;
-- Get the number of cons
        if (healthy0 1 = '1' and request0_0='0') then
        against := against + "0001";
            end if;
    if (healthy1 1 = '1' and request1_0 ='0') then
        against := against + "0001";
            end if;
    if (healthy2 1 = '1' and request2_0 ='0') then
        against := against + "0001";
            end if;
    if (healthy3 1 ='1' and request3_0 ='0') then
        against := against + "0001";
            end if;
    if (healthy4 1 ='1' and request4_0 ='0') then
        against := against + "0001";
            end if;
 -- final score
             if(pro = "0001" and against < "0001") then
      solution := '1';
           elsif(pro = "0010" and against < "0010") then
      solution := '1';
   elsif(pro = "0011" and against <  "0011") then
      solution := '1';
   elsif(pro = "0100" and against <  "0011") then
      solution := '1';
  elsif(pro = "0101" and against <  "0011") then
      solution := '1';

  else      solution := '0';
   end if;
            result <= solution;               -- put variable val into
            signal val
--          voteout_0 <= solution;            -- put variable val into
signal val

end process check_result;



result_latch:process(reset_0, clk_66_pal6)
begin
        IF (reset 0 = '0') THEN
                voteout 0 <= '1';
        ELSIF rising edge(clk 66 pal6) THEN
                IF result = '0' THEN
                        voteout_0 <= '0';

                END IF;
        END IF;
END PROCESS;

END voter;
```

mudlib.h 2/23/2001

```
/***************************************************************************
 *
 * FILENAME: mudlib.h
 *
 * CC NUMBER:
 *
 * ABSTRACT:
 *
 * USAGE:
 *
 * COMMENTS:
 *
 * AUTHOR: M. Vinskus
 *
 * DATE: 18-JUL-2000
 *
 ***************************************************************************
 **/

/* @MERCURY.COPYRIGHT.H@ */
```

mudlib.h 2/23/2001

```
#ifndef _MUDLIB_H
#define _MUDLIB_H

/*****************************************************************************
***
 *  INCLUDE FILES
 *****************************************************************************
 **/

#include <sal.h>

/*****************************************************************************
***
 *  DEFINED CONSTANTS
 *****************************************************************************
 **/

#define NUM_FINGERS_LOG  2
#define NUM_FINGERS_SQUARED_LOG  (2 * NUM_FINGERS_LOG)
#define NUM_FINGERS  (1 << NUM_FINGERS_LOG)
#define NUM_FINGERS_SQUARED  (1 << NUM_FINGERS_SQUARED_LOG)

#define L1_CACHE_SIZE  32768
#define L1_CACHE_LINE_SIZE  32

#define L1_CACHE_ALIGN_LOG  5
#define L1_CACHE_ALIGN  (1 << L1_CACHE_ALIGN_LOG)
#define L1_CACHE_ALIGN_MASK  (L1_CACHE_ALIGN - 1)

#define R_MATRIX_ALIGN_LOG  5
#define R_MATRIX_ALIGN  (1 << R_MATRIX_ALIGN_LOG)
#define R_MATRIX_ALIGN_MASK  (R_MATRIX_ALIGN - 1)

#define ALTIVEC_ALIGN_LOG  4
#define ALTIVEC_ALIGN  (1 << ALTIVEC_ALIGN_LOG)
#define ALTIVEC_ALIGN_MASK  (ALTIVEC_ALIGN - 1)

#define BF_CORR_FRAC_BITS 8
#define BF_CORR_FACTOR ((float)(1 << BF_CORR_FRAC_BITS))

#define BF_MPATH_FRAC_BITS 15                /* this should be dynamic */
#define BF_MPATH_FACTOR ((float)(1 << BF_MPATH_FRAC_BITS))

#define BF_RSUMS_FRAC_BITS ((2 * BF_MPATH_FRAC_BITS) - 16 +
BF_CORR_FRAC_BITS)
#define BF_RSUMS_FACTOR ((float)(1 << BF_RSUMS_FRAC_BITS))
#define BF_RSUMS_RFACTOR (1.0 / BF_RSUMS_FACTOR)

#define BF_RY_FRAC_BITS 9                /* 0 <= BF_RY_FRAC_BITS <= 14 */
#define BF_RY_FACTOR ((float)(1 << BF_RY_FRAC_BITS))
#define BF_RY_RFACTOR (1.0 / BF_RY_FACTOR)

#define BF_COMBINED_FACTOR ((float)( 1 <<
(BF_RSUMS_FRAC_BITS-BF_RY_FRAC_BITS)))
#define BF_COMBINED_RFACTOR (1.0 / BF_COMBINED_FACTOR)

#define BF8_ZERO 0
#define BF8_MAX 0x7f
#define BF8_RY_ONE ((BF8)(1 << BF_RY_FRAC_BITS))
#define BF16_RY_ONE ((BF16)(1 << BF_RY_FRAC_BITS))
#define BF16_RY_MONE (-BF16_RY_ONE)
#define BF16_ZERO 0
#define BF16_MAX 0x7fff
#define BF32_ZERO 0
#define BF32_RY_ONE ((BF32)(1 << BF_RY_FRAC_BITS))
#define BF32_MAX 0x7fffffff
```

```
mudlib.h                                                          2/23/2001

#define BIAS_8BIT 1

#define  BFABS( x )   (((x) >= 0) ? (x) : (-(x)))
#define  FABS( f )    (((f) >= 0.0) ? (f) : (-(f)))

/***************************************************************************
***
 *  TYPE DEFINITIONS
 ***************************************************************************
 **/
typedef long   BF32;
typedef short  BF16;
typedef char   BF8;

typedef struct {
  BF8  real;
  BF8  imag;
} COMPLEX_BF8;

typedef struct {
  BF16  real;
  BF16  imag;
} COMPLEX_BF16;

typedef struct {
  BF32  real;
  BF32  imag;
} COMPLEX_BF32;

/***************************************************************************
***
 *  MACRO DEFINITIONS
 ***************************************************************************
 **/
/* assumes (-(2.0 ^ 7) - 0.5) < (bf_factor * s) < ((2.0 ^ 7) - 0.5) */

#define SFtoBF8( bf_factor, s ) \
  ((BF8)((bf_factor) * (s) + (((s) > 0.0) ? 0.5 : -0.5)))

#define VFtoBF8( bf_factor, v, bfv, n ) \
  { \
    int i; \
    float factor = bf_factor; \
    vsmulx ( v, 1, &factor, v, 1, n, 0 ); \
    for ( i = 0; i < n; i++ ) \
      bfv[i] = (v[i] > 0.0) ? (BF8)(v[i] + 0.5) : (BF8)(v[i] - 0.5); \
  }

#define SBF8toF( bf_rfactor, bfs ) \
  ((bf_rfactor) * (float)(bfs))

#define VBF8toF( bf_rfactor, bfv, v, n ) \
  { \
    int i; \
    float rfactor = bf_rfactor; \
    for ( i = 0; i < n; i++ ) \
      v[i] = (float)bfv[i]; \
    vsmulx ( v, 1, &rfactor, v, 1, n, 0 ); \
  }

/* assumes (-(2.0 ^ 15) - 0.5) < (bf_factor * s) < ((2.0 ^ 15) - 0.5) */

#define SFtoBF16( bf_factor, s ) \
  ((BF16)((bf_factor) * (s) + (((s) > 0.0) ? 0.5 : -0.5)))

#define VFtoBF16( bf_factor, v, bfv, n ) \
  { \
```

```
    float factor = bf factor; \
    vsmulx ( (float *)v, 1, &factor, (float *)v, 1, n, 0 ); \
    vfixrx ( (float *)v, 1, (BF16 *)bfv, 1, n, 0 ); \
  }

#define SBF16toF( bf rfactor, bfs ) \
  ((bf_rfactor) * (float)(bfs))

#define VBF16toF( bf_rfactor, bfv, v, n ) \
  { \
    float rfactor = bf rfactor; \
    vfltx ( (short *)bfv, 1, v, 1, n, 0 ); \
    vsmulx ( v, 1, &rfactor, v, 1, n, 0 ); \
  }

/* assumes (-(2.0 ^ 31) - 0.5) < (bf_factor * x) < ((2.0 ^ 31) - 0.5) */

#define SFtoBF32( bf factor, s ) \
  ((BF32)((bf_factor) * (s) + (((s) > 0.0) ? 0.5 : -0.5)))

#define VFtoBF32( bf_factor, v, bfv, n ) \
  { \
    float factor = bf factor; \
    vsmulx ( v, 1, &factor, (float *)bfv, 1, n, 0 ); \
    vfixr32x ( (float *)bfv, 1, (int *)bfv, 1, n, 0 ); \
  }

#define SBF32toF( bf rfactor, bfs ) \
  ((bf_rfactor) * (float)(bfs))

#define VBF32toF( bf_rfactor, bfv, v, n ) \
  { \
    float rfactor = bf rfactor; \
    vflt32x ( (int *)bfv, 1, v, 1, n, 0 ); \
    vsmulx ( v, 1, &rfactor, v, 1, n, 0 ); \
  }

#define CORR SFtoBF( s )             SFtoBF8( BF CORR FACTOR, s )
#define MPATH_VFtoBF( v, bfv, n )  VFtoBF16( BF_MPATH_FACTOR, v, bfv, ((n)<<1)
)

#define BHAT SFtoBF( s )             ((BF8)((s) + (float)BIAS 8BIT))
#define BHAT SBFtoF( bfs )           ((float)(bfs) - (float)BIAS_8BIT)
#define BHAT_VFtoBF( v, bfv, n ) \
  { \
    float bias = (float)BIAS 8BIT; \
    vsaddx( v, 1, &bias, v, 1, n, 0 ); \
    fixpixax( v, 1, bfv, n, 0 ); \
  }

#define BHAT_VBFtoF( bfv, v, n ) \
  { \
    float bias = (float)(-BIAS 8BIT); \
    fltpixax( bfv, v, 1, n, 0 ); \
    vsaddx( v, 1, &bias, v, 1, n, 0 ); \
  }

#define RHAT SFtoBF( s )             SFtoBF8( BF RY FACTOR, s )
#define RHAT SBFtoF( bfs )           SBF8toF( BF RY RFACTOR, bfs )
#define RHAT VFtoBF( v, bfv, n )    VFtoBF8( BF RY FACTOR, v, bfv, n )
#define RHAT_VBFtoF( bfv, v, n )    VBF8toF( BF_RY_RFACTOR, bfv, v, n )

#define Y SFtoBF( s )                SFtoBF32( BF RY FACTOR, s )
#define Y SBFtoF( bfs )              SBF32toF( BF RY RFACTOR, bfs )
#define Y VFtoBF( v, bfv, n )       VFtoBF32( BF RY FACTOR, v, bfv, n )
#define Y_VBFtoF( bfv, v, n )       VBF32toF( BF_RY_RFACTOR, bfv, v, n )
```

mudlib.h 2/23/2001

```
#define MUDLIB_DECR_VIRT_USER( ptov_map, phys_user, virt_user ) \
  { \
    --virt user; \
    if ( virt user < 0 ) { \
      --phys user; \
      virt_user = ptov_map[phys_user] - 1; \
    } \
  }

#define MUDLIB_INCR_VIRT_USER( ptov_map, phys_user, virt_user ) \
  { \
    ++virt user; \
    if ( virt user == ptov_map[phys_user] ) { \
      ++phys user; \
      virt_user = 0; \
    } \
  }

/***********************************************************************
***
 *  PUBLIC FUNCTION PROTOTYPES
 ***********************************************************************
 **/

int  mudlib get Corr0 offset (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        );

int  mudlib get Corr0 size (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user,     /* must be < ptov_map[start_phys_user]
          */
          int  end phys user,       /* zero-based index into ptov map */
          int  end_virt_user        /* must be < ptov_map[end_phys_user] */
        );

int  mudlib get Corr1 offset (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        );

int  mudlib get Corr1 size (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user,     /* must be < ptov_map[start_phys_user]
          */
          int  end phys user,       /* zero-based index into ptov map */
          int  end_virt_user        /* must be < ptov_map[end_phys_user] */
        );
```

```
int  mudlib get R0 offset (
         unsigned char  *ptov_map, /* no more than 256 virts. per phys */
         int  tot_virt_users,      /* sum of ptov_map over all phys users
         */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user      /* must be < ptov_map[start_phys_user]
         */
       );

int  mudlib get R0 size (
         unsigned char  *ptov_map, /* no more than 256 virts. per phys */
         int  tot_virt_users,      /* sum of ptov_map over all phys users
         */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user,     /* must be < ptov_map[start_phys_user]
         */
         int  end phys user,       /* zero-based index into ptov map */
         int  end_virt_user        /* must be < ptov_map[end_phys_user] */
       );

int  mudlib get R1 offset (
         unsigned char  *ptov_map, /* no more than 256 virts. per phys */
         int  tot_virt_users,      /* sum of ptov_map over all phys users
         */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user      /* must be < ptov_map[start_phys_user]
         */
       );

int  mudlib get R1 size (
         unsigned char  *ptov_map, /* no more than 256 virts. per phys */
         int  tot_virt_users,      /* sum of ptov_map over all phys users
         */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user,     /* must be < ptov_map[start_phys_user]
         */
         int  end phys user,       /* zero-based index into ptov map */
         int  end_virt_user        /* must be < ptov_map[end_phys_user] */
       );

int  mudlib get num_virt_users (
         unsigned char  *ptov map, /* no more than 256 virts. per phys */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user,     /* must be < ptov_map[start_phys_user]
         */
         int  end phys user,       /* zero-based index into ptov map */
         int  end_virt_user        /* must be < ptov_map[end_phys_user] */
       );

void  mudlib get end user_pair (
         unsigned char  *ptov map, /* no more than 256 virts. per phys */
         int  start phys user,     /* zero-based index into ptov map */
         int  start_virt_user,     /* must be < ptov_map[start_phys_user]
         */
         int  num virt users,      /* number from start (must be > 0) */
         int  *end phys user,      /* zero-based index into ptov map */
         int  *end_virt_user       /* will be < ptov_map[*end_phys_user] */
       );

void  mudlib gen R (
         COMPLEX BF16  *mpath1 bf,
         COMPLEX BF16  *mpath2 bf,
         COMPLEX_BF8  *corr_0_bf,    /* adjusted for starting physical user
         */
         COMPLEX_BF8  *corr_1_bf,    /* adjusted for starting physical user
         */
```

```
          unsigned char  *ptov_map,    /* no more than 256 virts. per phys */
          float  *bf scalep,           /* scalar: always a power of 2 */
          float  *inv_scalep,          /* adjusted for starting physical user
          */
          float  *scalep,              /* start at 0'th physical user */
          char  *L1 cachep,
          BF8   *R0 upper bf,          /* must be 32-byte aligned */
          BF8   *R0 lower bf,
          BF8   *R1 trans_bf,
          BF8   *R1m bf,
          int   tot phys users,
          int   tot virt users,
          int   start phys user,       /* zero-based ("starting row") */
          int   start virt user,       /* relative to start phys user */
          int   end_phys_user,         /* actual number of "rows" to process
          */
          int   end_virt_user          /* relative to end_phys_user */
        );

void  mudlib 4R_to 3R (
          BF8   *R0 upper bf,          /* input matrix */
          BF8   *R0 lower bf,          /* input matrix */
          BF8   *R1 trans bf,          /* input matrix */
          char  *L1 cachep,            /* 32K-byte temp, 32-byte aligned */
          BF8   *R0 bf,                /* output matrix */
          BF8   *R1 bf,                /* output matrix */
          int   tot_virt_users
        );

void  mudlib_mpic ( BF8 *Bt hat,
                    BF8 *R0 hat,
                    BF8 *R1 hat,
                    BF8 *R1m_hat,
                    BF32 *Y,
                    BF32 Ythresh,
                    int N users,
                    int N bits,
                    int N_stages );

void  mudlib_reformat_corr ( COMPLEX  *in_corr,
                             COMPLEX BF8  *corr 0 bf,
                             COMPLEX BF8  *corr 1_bf,
                             int  num virt users,
                             int  num_multipath );

void  fixed_zidotprx ( COMPLEX SPLIT *A, int I, COMPLEX SPLIT *B, int J,
                       COMPLEX_SPLIT *C, int N, int X );

/*
 *  temp names (_v)
 */
int  mudlib get Corr0 offset v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start phys user,     /* zero-based index into ptov map */
          int  start_virt_user      /* must be < ptov_map[start_phys_user]
          */
        );

int  mudlib get Corr1 offset v (
          unsigned char  *ptov_map, /* no more than 256 virts. per phys */
          int  num fingers,         /* typically, 4 */
          int  tot_virt_users,      /* sum of ptov_map over all phys users
          */
          int  start_phys_user,     /* zero-based index into ptov_map */
```

```
            int   start_virt_user       /* must be < ptov_map[start_phys_user]
            */
        );

int  mudlib get R0 offset_v (
            unsigned char  *ptov_map,  /* no more than 256 virts. per phys */
            int   tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int   start phys user,     /* zero-based index into ptov map */
            int   start_virt_user      /* must be < ptov_map[start_phys_user]
            */
        );

int  mudlib get R0 size v (
            unsigned char  *ptov_map,  /* no more than 256 virts. per phys */
            int   tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int   start phys user,     /* zero-based index into ptov map */
            int   start_virt_user,     /* must be < ptov_map[start_phys_user]
            */
            int   end phys user,       /* zero-based index into ptov map */
            int   end_virt_user        /* must be < ptov_map[end_phys_user] */
        );

int  mudlib get R1 offset_v (
            unsigned char  *ptov_map,  /* no more than 256 virts. per phys */
            int   tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int   start phys user,     /* zero-based index into ptov map */
            int   start_virt_user      /* must be < ptov_map[start_phys_user]
            */
        );

int  mudlib get R1 size v (
            unsigned char  *ptov_map,  /* no more than 256 virts. per phys */
            int   tot_virt_users,      /* sum of ptov_map over all phys users
            */
            int   start phys user,     /* zero-based index into ptov map */
            int   start_virt_user,     /* must be < ptov_map[start_phys_user]
            */
            int   end phys user,       /* zero-based index into ptov map */
            int   end_virt_user        /* must be < ptov_map[end_phys_user] */
        );

#endif  /* _MUDLIB_H */
```

reformat_corr.c 2/23/2001

```
#include "mudlib.h"

#define INDEX_5D_TO_LIN(a0, a1, a2, a3, a4, max_a1, max_a2, max_a3, max_a4) \
             ((a4) + (max_a4) * ((a3) + (max_a3) * ((a2) + (max_a2) * ((a1)
             \
              + (max_a1) * (a0)))))

void  mudlib_reformat_corr (
          COMPLEX  *in_corr,
          COMPLEX_BF8  *corr_0_bf,
          COMPLEX_BF8  *corr_1_bf,
          int  num_virt_users,
          int  num_fingers )
{
  int  i, j, q, q1;

  for ( i = 0; i < num_virt_users; i++ ) {
    for ( j = (i+1); j < num_virt_users; j++ ) {
      for ( q = 0; q < num_fingers; q++ ) {
        for ( q1 = 0; q1 < num_fingers; q1++ ) {
          corr_0_bf->real = CORR_SFtoBF( in_corr[INDEX_5D_TO_LIN(
                                              0, i, j, q1, q,
                                              num_virt_users,
                                              num_virt_users,
                                              num_fingers,
                                              num_fingers)].real );
          corr_0_bf->imag = CORR_SFtoBF( in_corr[INDEX_5D_TO_LIN(
                                              0, i, j, q1, q,
                                              num_virt_users,
                                              num_virt_users,
                                              num_fingers,
                                              num_fingers)].imag );
          ++corr_0_bf;
        }
      }
    }
  }

  for ( i = 0; i < num_virt_users; i++ ) {
    for ( j = 0; j < num_virt_users; j++ ) {
      for ( q = 0; q < num_fingers; q++ ) {
        for ( q1 = 0; q1 < num_fingers; q1++ ) {
          corr_1_bf->real = CORR_SFtoBF( in_corr[INDEX_5D_TO_LIN(
                                              1, i, j, q1, q,
                                              num_virt_users,
                                              num_virt_users,
                                              num_fingers,
                                              num_fingers)].real );
          corr_1_bf->imag = CORR_SFtoBF( in_corr[INDEX_5D_TO_LIN(
                                              1, i, j, q1, q,
                                              num_virt_users,
                                              num_virt_users,
                                              num_fingers,
                                              num_fingers)].imag );
          ++corr_1_bf;
        }
      }
    }
  }
}
```

reformat_r.c 2/23/2001

```
#include "mudlib.h"

void mtrans32 8bit (
        BF8   *A,           /* logically contiguous input 32 x 32 blocks */
        BF8   *C,           /* output blocks separated by 32 * out_tc elements
        */
        char   *L1 cachep,
        int   A ncols,
        int   A nrows,
        int   C_tcols
      );

void mtriangle 8bit (
        BF8   *A,
        BF8   *C,
        int   N
      );

void  mudlib_4R to 3R (
        BF8   *R0 upper bf,                /* input matrix */
        BF8   *R0 lower bf,                /* input matrix */
        BF8   *R1 trans bf,                /* input matrix */
        char   *L1_cachep,                 /* temp: 32K bytes, 32-byte aligned
        */
        BF8   *R0 bf,                      /* output matrix */
        BF8   *R1 bf,                      /* output matrix */
        int   tot_virt_users
      )
{
  BF8  *R0 work;
  int  i, nrows, R0_tcols, tcols;

  tcols = (tot_virt_users + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;

  nrows = R_MATRIX ALIGN;
  for ( i = tot virt_users; i > 0; i -= R_MATRIX_ALIGN ) {
    if ( nrows > i ) nrows = i;
    mtrans32_8bit ( R1 trans bf, R1_bf, L1_cachep, tot_virt_users,
                    nrows, tcols );
    R1 trans_bf += (tcols << R_MATRIX_ALIGN_LOG);
    R1_bf += R_MATRIX_ALIGN;
  }

  R0 work = R0 bf;
  R0 tcols = tcols;
  nrows = R_MATRIX ALIGN;
  for ( i = tot virt_users; i > 0; i -= R_MATRIX_ALIGN ) {
    if ( nrows > i ) nrows = i;
    mtrans32 8bit ( R0 lower_bf, R0 work, L1 cachep, i, nrows, tcols );
    R0 lower bf += (R0_tcols << R MATRIX ALIGN LOG);
    R0 work += ((tcols << R MATRIX_ALIGN_LOG) + R_MATRIX_ALIGN);
    R0_tcols -= R_MATRIX_ALIGN;
  }

  mtriangle_8bit( R0_upper_bf, R0_bf, tot_virt_users );
}

#if COMPILE_C

void mtrans32 8bit (
        BF8   *A,          /* logically contiguous input A_nrows x A_ncols
        blocks */
        BF8   *C,          /* output blocks separated by 32 * C_tcols elements
        */
        char   *L1 cachep,
        int   A_ncols,
```

```
        int  A nrows,
        int  C_tcols
      )
{
  BF8  *Ap, *Cp;
  int  A tcols, C_nrows;
  int  i, j;

  (void)L1_cachep;

  A tcols = (A ncols + R MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;
  C_nrows = R_MATRIX_ALIGN;

  while ( A ncols ) {
    if ( A ncols < C_nrows ) C_nrows = A_ncols;
    Ap = A;
    Cp = C;
    for ( i = 0; i < A_nrows; i++ ) {
      for ( j = 0; j < C nrows; j++ )
        Cp[j * C tcols] = Ap[j];
      Ap += A tcols;
      Cp += 1;
    }
    A += R MATRIX_ALIGN;                        /* input travels horizontally */
    C += (C_tcols << R MATRIX_ALIGN_LOG);       /* output travels vertically */
    A_ncols -= C_nrows;
  }
}

void mtriangle 8bit (
        BF8  *A,
        BF8  *C,
        int  N
      )
{
  int  A counter, A_tcols, altivec_N, C_tcols;
  int  i, j;

  A counter = (N + R MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;
  C_tcols = A_counter + 1;

  altivec_N = (N + ALTIVEC_ALIGN_MASK) & ~ALTIVEC_ALIGN_MASK;

  for ( i = 0; i < N; i++ ) {
    for ( j = 0; j < altivec_N; j++ )
      C[j] = A[j];

    --altivec N;
    --A counter;
    A_tcols = (A counter + R_MATRIX_ALIGN_MASK) & ~R_MATRIX_ALIGN_MASK;
    A += (A tcols + 1);
    C += C_tcols;
  }
}

#endif                              /* COMPILE_C */
```

mtrans32_8bit.mac 2/23/2001

```
/*-----------------------------------------------------------------+
|   MC Standard Algorithms -- PPC Macro Language Version            |
|-------------------------------------------------------------------|
|                                                                   |
|  File Name:    mtrans32 8bit.mac                                  |
|  Description:  Perform N_tiles 32 x 32 byte transposes            |
|                                                                   |
|  void mtrans32 8bit (                                             |
|          BF8  *A,          contiguous input 32 x 32 blocks       |
|          BF8  *C,          output blocks separated by            |
|                            32 * out_tc elements                  |
|          char  *L1 cache,                                        |
|          int  A ncols,                                           |
|          int  A nrows,                                           |
|          int  C_tcols                                            |
|        )                                                          |
|                                                                   |
|  {                                                                |
|    BF8  *Ap, *Cp;                                                 |
|    int  A tcols, C_nrows;                                         |
|    int  i, j;                                                     |
|                                                                   |
|    A_tcols = (A ncols + R MATRIX ALIGN_MASK) &                    |
|              ~R MATRIX ALIGN_MASK;                                |
|    C_nrows = R_MATRIX_ALIGN;                                      |
|                                                                   |
|    while ( A ncols ) {                                            |
|      if ( A ncols < C_nrows ) C_nrows = A_ncols;                  |
|      Ap = A;                                                      |
|      Cp = C;                                                      |
|      for ( i = 0; i < A_nrows; i++ ) {                            |
|        for ( j = 0; j < C nrows; j++ )                            |
|          Cp[j * C tcols] = Ap[j];                                 |
|        Ap += A tcols;                                             |
|        Cp += 1;                                                   |
|      }                                                            |
|      A += R MATRIX_ALIGN;                                         |
|      C += (C_tcols << R MATRIX_ALIGN_LOG);                        |
|      A_ncols -= C_nrows;                                          |
|    }                                                              |
|  }                                                                |
|                                                                   |
|  Restrictions:  A, C and L1 cache must all be 16-byte aligned.    |
|                 C_tcols must be a multiple of 16.                 |
|                                                                   |
|            Mercury Computer Systems, Inc.                         |
|            Copyright (c) 2000  All rights reserved                |
|                                                                   |
|   Revision   Date    Engineer  Reason                             |
|   --------   ----    --------- -------                            |
|     0.0     000913   fpl       Created                            |
+-----------------------------------------------------------------*/

#include "salppc.inc"

#define DO_PREFETCH  1

#define LOAD INPUT( vT, rA, rB )        LVXL( vT, rA, rB )
#define LOAD_CACHE( vT, rA, rB )        LVX( vT, rA, rB )

#define STORE CACHE( vS, rA, rB )       STVX( vS, rA, rB )
#define STORE_OUTPUT( vS, rA, rB )      STVX( vS, rA, rB )

#define R MATRIX ALIGN_LOG    5
#define R MATRIX ALIGN        (1 << R MATRIX ALIGN_LOG)
#define R_MATRIX_ALIGN_MASK   (R_MATRIX_ALIGN - 1)
```

```
#define ALTIVEC ALIGN_LOG     4
#define ALTIVEC ALIGN         (1 << ALTIVEC ALIGN_LOG)
#define ALTIVEC_ALIGN_MASK    (ALTIVEC_ALIGN - 1)

#if DO PREFETCH
#define  PREFETCH( rA, rB, STRM, DST_BUMP ) \
   DSTT( rA, rB, STRM ) \
   ADD( rA, rA, DST_BUMP )
#else
#define  PREFETCH( rA, rB, STRM, DST_BUMP )
#endif

/**
 Four permute vectors for output stage
**/
RODATA_SECTION( 5 )

START_L_ARRAY( local_table )

L PERMUTE MASK( 0x00010405, 0x08090c0d, 0x10111415, 0x18191c1d )
L PERMUTE MASK( 0x02030607, 0x0a0b0e0f, 0x12131617, 0x1a1b1e1f )
L PERMUTE MASK( 0x00020406, 0x080a0c0e, 0x10121416, 0x181a1c1e )
L_PERMUTE_MASK( 0x01030507, 0x090b0d0f, 0x11131517, 0x191b1d1f )

END_ARRAY

/**
 Input parameters
**/
#define A          r3
#define C          r4
#define L1_cache   r5
#define NC         r6
#define NR         r7
#define TCC        r8

#define NC left    NC
#define TCA        r9
#define TCA4       r10
#define icount     r11

#define aptr0      r12
#define aptr1      r13
#define aptr2      r14
#define aptr3      r15

#define aindx0     r16
#define aindx1     r17
#define aindx2     r18
#define aindx3     r19

#define cptr0      r20
#define cptr1      r21
#define cptr2      r22
#define cptr3      r23

#define cindx0     r24
#define cindx1     r25
#define cindx2     r26
#define cindx3     r27

#define cindx4     aindx0
#define cindx5     aindx1
#define cindx6     aindx2
#define cindx7     aindx3
```

mtrans32_8bit.mac 2/23/2001

```
#define out_indx0  aptr0
#define out_indx1  aptr1
#define out_indx2  aptr2
#define out_indx3  aptr3

#define cptr      cptr0
#define outptr0   cptr1
#define outptr1   cptr2
#define TCC4      cptr3

#define tptr      icount
#define temp      aptr3

#define Cbump     r0
#define dstp      r0
#define dst_code  r28

/**
 G4 registers
**/
#define a00       v0
#define a01       v1
#define a02       v2
#define a03       v3

#define a10       v4
#define a11       v5
#define a12       v6
#define a13       v7

#define a20       v8
#define a21       v9
#define a22       v10
#define a23       v11

#define a30       v12
#define a31       v13
#define a32       v14
#define a33       v15

#define c00       v16
#define c01       v17
#define c02       v18
#define c03       v19

#define c10       v20
#define c11       v21
#define c12       v22
#define c13       v23

#define c20     c00
#define c21     c01
#define c22     c02
#define c23     c03

#define c30     c10
#define c31     c11
#define c32     c12
#define c33     c13

#define vt0       v24
#define vt1       v25
#define vt2       v26
#define vt3       v27

#define vt4     c00
#define vt5     c01
```

mtrans32_8bit.mac 2/23/2001

```
#define vt6     c02
#define vt7     c03

#define vp0       v28
#define vp1       v29
#define vp2       v30
#define vp3       v31

#define c0      a00
#define c1      a01
#define c2      a02
#define c3      a03
#define c4      a10
#define c5      a11
#define c6      a12
#define c7      a13

#define out0    a20
#define out1    a21
#define out2    a22
#define out3    a23
#define out4    a30
#define out5    a31
#define out6    a32
#define out7    a33

/**
 Text begins
**/
FUNC PROLOG
ENTRY_5( mtrans32_8bit, A, C, L1_cache, N, TCC )

  SAVE r13 r28
  USE_THRU_v31( VRSAVE_COND )

  ADDI( TCA, NC, R MATRIX_ALIGN_MASK )
  CMPWI( NC left, 32 )
  RLWINM( TCA, TCA, 0, 0, (31 - R_MATRIX_ALIGN_LOG) )

  LA( tptr, local table, 0 )
  MAKE_STREAM_CODE_IIR( dst_code, 64, 4, TCA )

    LVX( vp0, 0, tptr )
  ADDI( tptr, tptr, 16 )
    LVX( vp1, 0, tptr )
  ADDI( tptr, tptr, 16 )
  XORI( temp, A, 32 )
    LVX( vp2, 0, tptr )
  ADDI( tptr, tptr, 16 )
  SLWI( TCA4, TCA, 2 )
    LVX( vp3, 0, tptr )

  BLE( cont )

  ANDI C( temp, temp, 32 )
  BR( cont )

/**
 Outer loop transposes 2 (or 1 at end) 32 x 32 tiles per trip
**/
LABEL( outer_loop )
/* { */
   CMPWI( NC_left, 32 )

LABEL( cont )
   ADD( dstp, A, TCA4 )                    /* start prefetch advanced */
   MR( aptr0, A )
```

mtrans32_8bit.mac 2/23/2001

```
   ADD( dstp, dstp, TCA )              /* advanced further */
   LI( aindx0, 0 )
   ADD( aptr1, aptr0, TCA )
   LI( aindx1, 16 )
   ADD( aptr2, aptr1, TCA )
   MR( cptr0, L1 cache )
   ADD( aptr3, aptr2, TCA )

   ADDI( cptr1, cptr0, 512 )
   LI( cindx0, 0 )
    LOAD INPUT( a00, aptr0, aindx0 )   /*** begins next sequence ***/
   LI( cindx1, 128 )
    LOAD INPUT( a10, aptr1, aindx0 )
   LI( cindx2, 256 )
    LOAD INPUT( a20, aptr2, aindx0 )
   LI( cindx3, 384 )
    LOAD INPUT( a30, aptr3, aindx0 )
   MR( icount, NR )

   BLE( input_loop_do1 )

   LI( aindx2, 32 )               /* these are used only in two tile loop */
     LOAD INPUT( a02, aptr0, aindx2 )
   LI( aindx3, 48 )
     LOAD INPUT( a12, aptr1, aindx2 )
   ADDI( cptr2, cptr1, 512 )
     LOAD INPUT( a22, aptr2, aindx2 )
   ADDI( cptr3, cptr2, 512 )
     LOAD_INPUT( a32, aptr3, aindx2 )

/**
 Top of input loop processes a 4 x 64 byte tile each trip
**/
LABEL( input_loop_do2 )
/* { */
  PREFETCH( dstp, dst code, 0, TCA4 )
    ADDIC C( icount, icount, -4 )
      VMRGHW(vt0, a00, a20)   /* vt0 = a00[0-3] a20[0-3] a00[4-7] a20[4-7] */
    LOAD INPUT( a01, aptr0, aindx1 )
      VMRGLW(vt2, a00, a20)   /* vt2 = a00[8-b] a20[8-b] a00[c-f] a20[c-f] */
    LOAD INPUT( a11, aptr1, aindx1 )
      VMRGHW(vt1, a10, a30)   /* vt1 = a10[0-3] a30[0-3] a10[4-7] a30[4-7] */
    LOAD INPUT( a21, aptr2, aindx1 )
      VMRGLW(vt3, a10, a30)   /* vt3 = a10[8-b] a10[8-b] a30[c-f] a30[c-f] */
    LOAD_INPUT( a31, aptr3, aindx1 )

      VMRGHW(c00, vt0, vt1)   /* c00 = a00[0-3] a10[0-3] a20[0-3] a30[0-3] */
    STORE CACHE( c00, cptr0, cindx0 )
      VMRGLW(c01, vt0, vt1)   /* c01 = a00[4-7] a10[4-7] a20[4-7] a30[4-7] */
    STORE CACHE( c01, cptr0, cindx1 )
      VMRGHW(c02, vt2, vt3)   /* c02 = a00[8-b] a10[8-b] a20[8-b] a30[8-b] */
    STORE CACHE( c02, cptr0, cindx2 )
      VMRGLW(c03, vt2, vt3)   /* c03 = a00[c-f] a10[c-f] a20[c-f] a30[c-f] */
    STORE_CACHE( c03, cptr0, cindx3 )

      VMRGHW(vt0, a01, a21)   /* vt0 = a01[0-3] a21[0-3] a01[4-7] a21[4-7] */
    LOAD INPUT( a03, aptr0, aindx3 )
      VMRGLW(vt2, a01, a21)   /* vt2 = a01[8-b] a21[8-b] a01[c-f] a21[c-f] */
    LOAD INPUT( a13, aptr1, aindx3 )
      VMRGHW(vt1, a11, a31)   /* vt1 = a11[0-3] a31[0-3] a11[4-7] a31[4-7] */
    LOAD INPUT( a23, aptr2, aindx3 )
      VMRGLW(vt3, a11, a31)   /* vt3 = a11[8-b] a11[8-b] a31[c-f] a31[c-f] */
    LOAD_INPUT( a33, aptr3, aindx3 )

      VMRGHW(c10, vt0, vt1)   /* c10 = a01[0-3] a11[0-3] a21[0-3] a31[0-3] */
    STORE CACHE( c10, cptr1, cindx0 )
      VMRGLW(c11, vt0, vt1)   /* c11 = a01[4-7] a11[4-7] a21[4-7] a31[4-7] */
```

```
    STORE CACHE( c11, cptr1, cindx1 )
      VMRGHW(c12, vt2, vt3)   /* c12 = a01[8-b] a11[8-b] a21[8-b] a31[8-b] */
    STORE CACHE( c12, cptr1, cindx2 )
      VMRGLW(c13, vt2, vt3)   /* c13 = a01[c-f] a11[c-f] a21[c-f] a31[c-f] */
    STORE_CACHE( c13, cptr1, cindx3 )

  BLE( flush_input_loop_do2 )

  ADD( aindx0, aindx0, TCA4 )    /* bump for next load sequence */
  ADD( aindx1, aindx1, TCA4 )
  ADD( aindx2, aindx2, TCA4 )
  ADD( aindx3, aindx3, TCA4 )

      VMRGHW(vt0, a02, a22)   /* vt0 = a02[0-3] a22[0-3] a02[4-7] a22[4-7] */
    LOAD INPUT( a00, aptr0, aindx0 )    /*** begins next sequence ***/
      VMRGLW(vt2, a02, a22)   /* vt2 = a02[8-b] a22[8-b] a02[c-f] a22[c-f] */
    LOAD INPUT( a02, aptr0, aindx2 )
      VMRGHW(vt1, a12, a32)   /* vt1 = a12[0-3] a32[0-3] a12[4-7] a32[4-7] */
    LOAD INPUT( a10, aptr1, aindx0 )
      VMRGLW(vt3, a12, a32)   /* vt3 = a12[8-b] a12[8-b] a32[c-f] a32[c-f] */
    LOAD_INPUT( a12, aptr1, aindx2 )

      VMRGHW(c20, vt0, vt1)   /* c20 = a02[0-3] a12[0-3] a22[0-3] a32[0-3] */
    STORE CACHE( c20, cptr2, cindx0 )
      VMRGLW(c21, vt0, vt1)   /* c21 = a02[4-7] a12[4-7] a22[4-7] a32[4-7] */
    STORE CACHE( c21, cptr2, cindx1 )
      VMRGHW(c22, vt2, vt3)   /* c22 = a02[8-b] a12[8-b] a22[8-b] a32[8-b] */
    STORE CACHE( c22, cptr2, cindx2 )
      VMRGLW(c23, vt2, vt3)   /* c23 = a02[c-f] a12[c-f] a22[c-f] a32[c-f] */
    STORE_CACHE( c23, cptr2, cindx3 )

      VMRGHW(vt0, a03, a23)   /* vt0 = a03[0-3] a23[0-3] a03[4-7] a23[4-7] */
    LOAD INPUT( a20, aptr2, aindx0 )
      VMRGLW(vt2, a03, a23)   /* vt2 = a03[8-b] a23[8-b] a03[c-f] a23[c-f] */
    LOAD INPUT( a22, aptr2, aindx2 )
      VMRGHW(vt1, a13, a33)   /* vt1 = a13[0-3] a33[0-3] a13[4-7] a33[4-7] */
    LOAD INPUT( a30, aptr3, aindx0 )
      VMRGLW(vt3, a13, a33)   /* vt3 = a13[8-b] a13[8-b] a33[c-f] a33[c-f] */
    LOAD_INPUT( a32, aptr3, aindx2 )

      VMRGHW(c30, vt0, vt1)   /* c30 = a03[0-3] a13[0-3] a23[0-3] a33[0-3] */
    STORE CACHE( c30, cptr3, cindx0 )
      VMRGLW(c31, vt0, vt1)   /* c31 = a03[4-7] a13[4-7] a23[4-7] a33[4-7] */
    STORE CACHE( c31, cptr3, cindx1 )
      VMRGHW(c32, vt2, vt3)   /* c32 = a03[8-b] a13[8-b] a23[8-b] a33[8-b] */
    STORE CACHE( c32, cptr3, cindx2 )
      VMRGLW(c33, vt2, vt3)   /* c33 = a03[c-f] a13[c-f] a23[c-f] a33[c-f] */
    STORE_CACHE( c33, cptr3, cindx3 )

  ADDI( cindx0, cindx0, 16 )     /* bump for next store sequence */
  ADDI( cindx1, cindx1, 16 )
  ADDI( cindx2, cindx2, 16 )
  ADDI( cindx3, cindx3, 16 )

  BR( input_loop_do2 )

LABEL( flush_input_loop_do2 )

      VMRGHW(vt0, a02, a22)   /* vt0 = a02[0-3] a22[0-3] a02[4-7] a22[4-7] */
      VMRGLW(vt2, a02, a22)   /* vt2 = a02[8-b] a22[8-b] a02[c-f] a22[c-f] */
      VMRGHW(vt1, a12, a32)   /* vt1 = a12[0-3] a32[0-3] a12[4-7] a32[4-7] */
      VMRGLW(vt3, a12, a32)   /* vt3 = a12[8-b] a12[8-b] a32[c-f] a32[c-f] */

      VMRGHW(c20, vt0, vt1)   /* c20 = a02[0-3] a12[0-3] a22[0-3] a32[0-3] */
    STORE CACHE( c20, cptr2, cindx0 )
      VMRGLW(c21, vt0, vt1)   /* c21 = a02[4-7] a12[4-7] a22[4-7] a32[4-7] */
    STORE_CACHE( c21, cptr2, cindx1 )
```

mtrans32_8bit.mac 2/23/2001

```
      VMRGHW(c22, vt2, vt3)    /* c22 = a02[8-b] a12[8-b] a22[8-b] a32[8-b] */
    STORE CACHE( c22, cptr2, cindx2 )
      VMRGLW(c23, vt2, vt3)    /* c23 = a02[c-f] a12[c-f] a22[c-f] a32[c-f] */
    STORE_CACHE( c23, cptr2, cindx3 )

      VMRGHW(vt0, a03, a23)    /* vt0 = a03[0-3] a23[0-3] a03[4-7] a23[4-7] */
      VMRGLW(vt2, a03, a23)    /* vt2 = a03[8-b] a23[8-b] a03[c-f] a23[c-f] */
      VMRGHW(vt1, a13, a33)    /* vt1 = a13[0-3] a33[0-3] a13[4-7] a33[4-7] */
      VMRGLW(vt3, a13, a33)    /* vt3 = a13[8-b] a13[8-b] a33[c-f] a33[c-f] */

      VMRGHW(c30, vt0, vt1)    /* c30 = a03[0-3] a13[0-3] a23[0-3] a33[0-3] */
    STORE CACHE( c30, cptr3, cindx0 )
      VMRGLW(c31, vt0, vt1)    /* c31 = a03[4-7] a13[4-7] a23[4-7] a33[4-7] */
    STORE CACHE( c31, cptr3, cindx1 )
      VMRGHW(c32, vt2, vt3)    /* c32 = a03[8-b] a13[8-b] a23[8-b] a33[8-b] */
    STORE CACHE( c32, cptr3, cindx2 )
      VMRGLW(c33, vt2, vt3)    /* c33 = a03[c-f] a13[c-f] a23[c-f] a33[c-f] */
    STORE_CACHE( c33, cptr3, cindx3 )

  MR( outptr0, C )              /* set for output loop in current pass */
  SLWI( Cbump, TCC, 6 )
  ADDI( A, A, 64 )
  ADD( C, C, Cbump )            /* bump C for next pass */
  LI( icount, 64 )              /* set icount for 2 tiles */
  BR( output_start )            /* join to common output loop */

/**
 Top of input loop processes a 4 x 32 byte tile each trip
**/
LABEL( input_loop_do1 )
/* { */
  PREFETCH( dstp, dst code, 0, TCA4 )
    ADDIC C( icount, icount, -4 )
      VMRGHW(vt0, a00, a20)    /* vt0 = a00[0-3] a20[0-3] a00[4-7] a20[4-7] */
    LOAD INPUT( a01, aptr0, aindx1 )
      VMRGLW(vt2, a00, a20)    /* vt2 = a00[8-b] a20[8-b] a00[c-f] a20[c-f] */
    LOAD INPUT( a11, aptr1, aindx1 )
      VMRGHW(vt1, a10, a30)    /* vt1 = a10[0-3] a30[0-3] a10[4-7] a30[4-7] */
    LOAD INPUT( a21, aptr2, aindx1 )
      VMRGLW(vt3, a10, a30)    /* vt3 = a10[8-b] a10[8-b] a30[c-f] a30[c-f] */
    LOAD_INPUT( a31, aptr3, aindx1 )

      VMRGHW(c00, vt0, vt1)    /* c00 = a00[0-3] a10[0-3] a20[0-3] a30[0-3] */
    STORE CACHE( c00, cptr0, cindx0 )
      VMRGLW(c01, vt0, vt1)    /* c01 = a00[4-7] a10[4-7] a20[4-7] a30[4-7] */
    STORE CACHE( c01, cptr0, cindx1 )
      VMRGHW(c02, vt2, vt3)    /* c02 = a00[8-b] a10[8-b] a20[8-b] a30[8-b] */
    STORE CACHE( c02, cptr0, cindx2 )
      VMRGLW(c03, vt2, vt3)    /* c03 = a00[c-f] a10[c-f] a20[c-f] a30[c-f] */
    STORE_CACHE( c03, cptr0, cindx3 )

  BLE( flush_input_loop_do1 )

  ADD( aindx0, aindx0, TCA4 )   /* bump for next load sequence */
  ADD( aindx1, aindx1, TCA4 )

      VMRGHW(vt0, a01, a21)    /* vt0 = a01[0-3] a21[0-3] a01[4-7] a21[4-7] */
    LOAD INPUT( a00, aptr0, aindx0 )    /*** begins next sequence ***/
      VMRGLW(vt2, a01, a21)    /* vt2 = a01[8-b] a21[8-b] a01[c-f] a21[c-f] */
    LOAD INPUT( a10, aptr1, aindx0 )
      VMRGHW(vt1, a11, a31)    /* vt1 = a11[0-3] a31[0-3] a11[4-7] a31[4-7] */
    LOAD INPUT( a20, aptr2, aindx0 )
      VMRGLW(vt3, a11, a31)    /* vt3 = a11[8-b] a11[8-b] a31[c-f] a31[c-f] */
    LOAD_INPUT( a30, aptr3, aindx0 )

      VMRGHW(c10, vt0, vt1)    /* c10 = a01[0-3] a11[0-3] a21[0-3] a31[0-3] */
    STORE_CACHE( c10, cptr1, cindx0 )
```

mtrans32_8bit.mac 2/23/2001

```
      VMRGLW(c11, vt0, vt1)   /* c11 = a01[4-7] a11[4-7] a21[4-7] a31[4-7] */
    STORE CACHE( c11, cptr1, cindx1 )
      VMRGHW(c12, vt2, vt3)   /* c12 = a01[8-b] a11[8-b] a21[8-b] a31[8-b] */
    STORE CACHE( c12, cptr1, cindx2 )
      VMRGLW(c13, vt2, vt3)   /* c13 = a01[c-f] a11[c-f] a21[c-f] a31[c-f] */
    STORE_CACHE( c13, cptr1, cindx3 )

  ADDI( cindx0, cindx0, 16 )    /* bump for next store sequence */
  ADDI( cindx1, cindx1, 16 )
  ADDI( cindx2, cindx2, 16 )
  ADDI( cindx3, cindx3, 16 )

  BR( input_loop_do1 )

LABEL( flush_input_loop_do1 )

      VMRGHW(vt0, a01, a21)   /* vt0 = a01[0-3] a21[0-3] a01[4-7] a21[4-7] */
      VMRGLW(vt2, a01, a21)   /* vt2 = a01[8-b] a21[8-b] a01[c-f] a21[c-f] */
      VMRGHW(vt1, a11, a31)   /* vt1 = a11[0-3] a31[0-3] a11[4-7] a31[4-7] */
      VMRGLW(vt3, a11, a31)   /* vt3 = a11[8-b] a11[8-b] a31[c-f] a31[c-f] */

      VMRGHW(c10, vt0, vt1)   /* c10 = a01[0-3] a11[0-3] a21[0-3] a31[0-3] */
    STORE CACHE( c10, cptr1, cindx0 )
      VMRGLW(c11, vt0, vt1)   /* c11 = a01[4-7] a11[4-7] a21[4-7] a31[4-7] */
    STORE CACHE( c11, cptr1, cindx1 )
      VMRGHW(c12, vt2, vt3)   /* c12 = a01[8-b] a11[8-b] a21[8-b] a31[8-b] */
    STORE CACHE( c12, cptr1, cindx2 )
      VMRGLW(c13, vt2, vt3)   /* c13 = a01[c-f] a11[c-f] a21[c-f] a31[c-f] */
    STORE_CACHE( c13, cptr1, cindx3 )

  MR( outptr0, C )              /* set for output loop in current pass */
  SLWI( Cbump, TCC, 5 )
  ADDI( A, A, 32 )
  ADD( C, C, Cbump )            /* bump C for next pass */
  LI( icount, 32 )              /* set icount for 1 tile */

/**
 Second stage of transposition, write output
**/
LABEL( output_start )

  CMPW_CR( 6, icount, NC_left )

  MR( cptr, L1 cache )
  SLWI( TCC4, TCC, 2 )
  LI( cindx0, 0 )
  LI( cindx1, 16 )
  LI( cindx2, 2*16 )
  LI( cindx3, 3*16 )
  LI( cindx4, 4*16 )
  LI( cindx5, 5*16 )
  LI( cindx6, 6*16 )

  BLE_CR( 6, PC OFFSET( 8 ) )
  MR( icount, NC_left )

  LI( cindx7, 7*16 )
  SUB( NC_left, NC_left, icount )

  ADDIC C( icount, icount, -4 )
  LI( out indx0, 0 )
    LOAD CACHE( c0, cptr, cindx0 )
  ADD( out indx1, out indx0, TCC )
    LOAD CACHE( c1, cptr, cindx1 )
  ADD( out indx2, out indx1, TCC )
    LOAD CACHE( c2, cptr, cindx2 )
  ADD( out_indx3, out_indx2, TCC )
```

mtrans32_8bit.mac 2/23/2001

```
    LOAD CACHE( c3, cptr, cindx3 )
  ADDI( outptr1, outptr0, 16 )
    LOAD CACHE( c4, cptr, cindx4 )
      VPERM( vt0, c0, c1, vp0 )
    LOAD CACHE( c5, cptr, cindx5 )
      VPERM( vt1, c0, c1, vp1 )
    LOAD CACHE( c6, cptr, cindx6 )
      VPERM( vt2, c2, c3, vp0 )
    LOAD CACHE( c7, cptr, cindx7 )
      VPERM( vt3, c2, c3, vp1 )
  ADDI( cptr, cptr, 128 )
  BR( output_mloop )

/**
 Loop outputs four 32 byte rows
**/
LABEL( output loop )
  ADDIC_C( icount, icount, -4 )
  ADDI( cptr, cptr, 128 )

    STORE OUTPUT( out0, outptr0, out_indx0 )
      VPERM( out4, vt4, vt6, vp2 )
    STORE OUTPUT( out4, outptr1, out_indx0 )
      VPERM( out5, vt4, vt6, vp3 )
    STORE OUTPUT( out1, outptr0, out_indx1 )
      VPERM( out6, vt5, vt7, vp2 )
    STORE OUTPUT( out5, outptr1, out_indx1 )
      VPERM( out7, vt5, vt7, vp3 )

    STORE OUTPUT( out2, outptr0, out_indx2 )
      VPERM( vt0, c0, c1, vp0 )
    STORE OUTPUT( out6, outptr1, out_indx2 )
      VPERM( vt1, c0, c1, vp1 )
    STORE OUTPUT( out3, outptr0, out_indx3 )
      VPERM( vt2, c2, c3, vp0 )
    STORE OUTPUT( out7, outptr1, out_indx3 )
      VPERM( vt3, c2, c3, vp1 )

    ADD( outptr0, outptr0, TCC4 )
    ADD( outptr1, outptr1, TCC4 )

LABEL( output mloop )
  BLE( flush output_loop )
    LOAD CACHE( c0, cptr, cindx0 )
      VPERM( vt4, c4, c5, vp0 )
    LOAD CACHE( c1, cptr, cindx1 )
      VPERM( vt5, c4, c5, vp1 )
    LOAD CACHE( c2, cptr, cindx2 )
      VPERM( vt6, c6, c7, vp0 )
    LOAD CACHE( c3, cptr, cindx3 )
      VPERM( vt7, c6, c7, vp1 )

    LOAD CACHE( c4, cptr, cindx4 )
      VPERM( out0, vt0, vt2, vp2 )
    LOAD CACHE( c5, cptr, cindx5 )
      VPERM( out1, vt0, vt2, vp3 )
    LOAD CACHE( c6, cptr, cindx6 )
      VPERM( out2, vt1, vt3, vp2 )
    LOAD CACHE( c7, cptr, cindx7 )
      VPERM( out3, vt1, vt3, vp3 )

    BR( output_loop )

LABEL( flush_output_loop )

      VPERM( vt4, c4, c5, vp0 )
      VPERM( vt5, c4, c5, vp1 )
```

```
      VPERM( vt6, c6, c7, vp0 )
      VPERM( vt7, c6, c7, vp1 )

  CMPWI( icount, -3 )
      VPERM( out0, vt0, vt2, vp2 )
    STORE OUTPUT( out0, outptr0, out_indx0 )
      VPERM( out4, vt4, vt6, vp2 )
    STORE OUTPUT( out4, outptr1, out_indx0 )
  BEQ( oloop_next )

  CMPWI( icount, -2 )
      VPERM( out1, vt0, vt2, vp3 )
    STORE OUTPUT( out1, outptr0, out_indx1 )
      VPERM( out5, vt4, vt6, vp3 )
    STORE OUTPUT( out5, outptr1, out_indx1 )
  BEQ( oloop_next )

  CMPWI( icount, -1 )
      VPERM( out2, vt1, vt3, vp2 )
    STORE OUTPUT( out2, outptr0, out_indx2 )
      VPERM( out6, vt5, vt7, vp2 )
    STORE OUTPUT( out6, outptr1, out_indx2 )
  BEQ( oloop_next )

      VPERM( out3, vt1, vt3, vp3 )
    STORE OUTPUT( out3, outptr0, out_indx3 )
      VPERM( out7, vt5, vt7, vp3 )
    STORE_OUTPUT( out7, outptr1, out_indx3 )

/**
 Next four rows of C?
**/
LABEL( oloop next )
   BLT_CR( 6, outer_loop )            /* branch if icount < NC_left */

/**
 Exit routine
**/
 LABEL( ret )
 FREE THRU v31( VRSAVE_COND )
 REST r13_r28
 RETURN

FUNC_EPILOG
```

mtriangle_8bit.mac 2/23/2001

```
/*---------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version    ---|
|----------------------------------------------------------------|
|                                                                |
|  File Name:     mtriangle_8bit.mac.                            |
|  Description:   Move from an upper triangular matrix stored    |
|                 as a series of 32-line rectangles, each of     |
|                 width 32 elements less than its immediate      |
|                 predecessor to the upper triangle of an        |
|                 full N x N matrix.                             |
|                                                                |
|  mtriangle_8bit ( char *A, char *C, int N )                    |
|                                                                |
|  Restrictions:  A, B and C must all be 16-byte aligned.        |
|                 N must be a multiple of 16 and >= 16.          |
|                                                                |
|            Mercury Computer Systems, Inc.                      |
|            Copyright (c) 2000  All rights reserved             |
|                                                                |
|   Revision      Date     Engineer  Reason                      |
|   --------      ----     -------- --------                     |
|     0.0        000605      jg    Created                       |
+--------------------------------------------------------------*/

#include "salppc.inc"

#define LOAD_A( vT, rA, rB )       LVXL( vT, rA, rB )
#define LOAD_C( vT, rA, rB )       LVX( vT, rA, rB )
#define STORE_C( vS, rA, rB )      STVX( vS, rA, rB )

#define R_MATRIX_ALIGN_LOG    5
#define R_MATRIX_ALIGN        (1 << R_MATRIX_ALIGN_LOG)
#define R_MATRIX_ALIGN_MASK   (R_MATRIX_ALIGN - 1)

#define ALTIVEC_ALIGN_LOG     4
#define ALTIVEC_ALIGN         (1 << ALTIVEC_ALIGN_LOG)
#define ALTIVEC_ALIGN_MASK    (ALTIVEC_ALIGN - 1)

/**
 Input parameters
**/
#define A              r3
#define C              r4
#define N              r5

#define A_tcols        r6
#define C_tcols        r7
#define altivec_N      r8
#define A_counter      r9
#define index0         r10
#define index1         r11
#define index2         r12
#define index3         r13

#define count          r0

#define a0             v0
#define a1             v1
#define a2             v2
#define a3             v3
#define c0             v4
#define shift          v5
#define shift_incr     v6
#define mask           v7
#define left           v8
#define right          v9
```

mtriangle_8bit.mac 2/23/2001

```
FUNC_PROLOG

ENTRY_3( mtriangle_8bit, A, C, N )

  SAVE r13
  USE_THRU_v9( VRSAVE_COND )

  ADDI( A counter, N, R MATRIX ALIGN_MASK )
      VSPLTISW( shift_incr, 8 )
  ADDI( altivec N, N, ALTIVEC ALIGN_MASK )
        VXOR( shift, shift, shift )
  RLWINM( A counter, A counter, 0, 0, (31 - R MATRIX ALIGN LOG) )
  RLWINM( altivec N, altivec N, 0, 0, (31 - ALTIVEC_ALIGN_LOG ) )
  ADDI( C_tcols, A_counter, 1 )

LABEL( oloop )

  ADDIC C( count, altivec_N, -64 )
    LOAD C( c0, 0, C )
      VSPLTISW( mask, -1 )
    LOAD A( a0, 0, A )
      VSRO( mask, mask, shift )
  LI( index0, 16 )
        VANDC( left, c0, mask )
  LI( index1, 32 )
        VAND( right, a0, mask )
  LI( index2, 48 )
        VOR( c0, left, right )
    STORE C( c0, 0, C )
  BLE( dosmall )
  LI( index3, 64 )

LABEL( iloop )
    LOAD A( a0, A, index0 )
  ADDIC C( count, count, -64 )
    LOAD A( a1, A, index1 )
    LOAD A( a2, A, index2 )
    LOAD A( a3, A, index3 )
    STORE C( a0, C, index0 )
  ADDI( index0, index0, 64 )
    STORE C( a1, C, index1 )
  ADDI( index1, index1, 64 )
    STORE C( a2, C, index2 )
  ADDI( index2, index2, 64 )
    STORE C( a3, C, index3 )
  ADDI( index3, index3, 64 )
  BGT( iloop )

LABEL( dosmall )
  ADDIC C( count, count, 48 )
  BLE( windout )

LABEL( sloop )
  ADDIC C( count, count, -16 )
    LOAD A( a0, A, index0 )
    STORE C( a0, C, index0 )
  ADDI( index0, index0, 16 )
  BGT( sloop )

LABEL( windout )
  DECR_C( N )
        VADDUWM( shift, shift, shift_incr )
  ADDI( A counter, A_counter, -1 )
  ADDI( A, A, 1 )
  ADDI( A tcols, A counter, R_MATRIX_ALIGN_MASK )
  DECR( altivec_N )
```

mtriangle_8bit.mac 2/23/2001

```
  RLWINM( A_tcols, A_tcols, 0, 0, (31 - R_MATRIX_ALIGN_LOG) )
  ADD( C, C, C_tcols )
  ADD( A, A, A_tcols )
  BNE( oloop )

  FREE_THRU_v9( VRSAVE_COND )
  REST r13
  RETURN

FUNC_EPILOG
```

salppc.h 2/23/2001

```
#if !defined( SALPPC_H )
#define SALPPC_H

#if 0
+****************************************************************************+
|***   MC Standard Algorithms -- PPC Version                             ***|
*****************************************************************************|
|*                                                                         *|
|*        File Name:       salppc.h                                        *|
|*        Description:     SAL macro include file                          *|
|*                                                                         *|
|*        Source files should have extension .mac. For example, vadd.mac   *|
|*        and must include this file (salppc.h).                           *|
|*                                                                         *|
|*        To assemble for PPC ucode, use the following basic               *|
|*        makefile build rule:                                             *|
|*                                                                         *|
|*           .SUFFIXES: .mac .c .s .o                                      *|
|*                                                                         *|
|*           .mac.o:                                                       *|
|*              cp $*.mac $*.c                                             *|
|*              ccmc -o $*.s -E $*.c                                       *|
|*              ccmc -c -o $*.o $*.s                                       *|
|*              rm -f $*.s                                                 *|
|*              rm -f $*.c                                                 *|
|*                                                                         *|
|*        To compile for C, use the following basic makefile build rule:   *|
|*                                                                         *|
|*           .SUFFIXES: .mac .c .o                                         *|
|*                                                                         *|
|*           .mac.o:                                                       *|
|*              cp $*.mac $*.c                                             *|
|*              ccmc -DCOMPILE_C -c -o $*.o $*.c                           *|
|*              rm -f $*.c                                                 *|
|*                                                                         *|
|*        The first 8 function arguments are passed in GPR registers       *|
|*        r3 - r10. Arguments beyond 8 are passed on the stack and may     *|
|*        be obtained with the GET_ARG8, GET_ARG9, ... GET ARG15 macros.   *|
|*        Additional GPR registers should be assigned in ascending order   *|
|*        starting from the last function argument. These may be declared *|
|*        with the DECLARE_rx[ ry] macros. For example, a function with    *|
|*        5 arguments that requires 3 additional GPR registers would       *|
|*        issue: DECLARE r8 r10. r0, if required, should be declared       *|
|*        separately with the DECLARE r0 macro. GPR registers above r12    *|
|*        must be saved and restored using the SAVE_r13[_ry] and           *|
|*        REST_r13[_ry] macros, respectively.                              *|
|*                                                                         *|
|*        FPR registers should be assigned in ascending order starting     *|
|*        with f0[d0]. These may be declared with the DECLARE_f0[_fy]      *|
|*        or DECLARE d0[ dy] macros.                                       *|
|*        For example, DECLARE f0 f11. FPR registers above f13[d13] must   *|
|*        be saved and restored using the SAVE f14[ fy] and REST f14[_fy]  *|
|*        or SAVE_d14[_dy] and REST_d14[_dy] macros, respectively.         *|
|*                                                                         *|
|*        All variables must be assigned a register using the              *|
|*        pre-processor #define directive. GPR registers are named         *|
|*        r0 - r31; Single precision FPR registers are named f0 - f31.     *|
|*        Double precision FPR registers are named d0 - d31. Different     *|
|*        variables may be assigned to the same register as in:            *|
|*                                                                         *|
|*           #define vara  f12                                             *|
|*           #define varb  f12                                             *|
|*                                                                         *|
|*        Functions must begin with the FUNC_PROLOG macro and end          *|
|*        with the FUNC_EPILOG macro.                                      *|
|*                                                                         *|
```

salppc.h 2/23/2001

```
|*        Macros are provided for both Fortran and C entry points.       *|
|*                                                                        *|
|*        The GET SALCACHE macro should be used to get the address of    *|
|*        the "current" salcache buffer into a GPR register.              *|
|*                                                                        *|
|*        Avoid terminating macro lines with a semicolon.                 *|
|*                                                                        *|
|*        The following example demonstrates typical usage:               *|
|*                                                                        *|
|*           #include "salppc.h"                                          *|
|*                                                                        *|
|*           /*                                                           *|
|*            *  assign variables to registers                            *|
|*            */                                                          *|
|*           #define A  r3                                                *|
|*           #define I  r4                                                *|
|*           #define B  r5                                                *|
|*           #define J  r6                                                *|
|*           #define C  r7                                                *|
|*           #define K  r8                                                *|
|*           #define D  r9                                                *|
|*           #define L  r10                                               *|
|*           #define N  r12                                               *|
|*           #define EFLAG  r11                                           *|
|*           #define count  r11                                           *|
|*                                                                        *|
|*           #define t0  r13                                              *|
|*           #define t1  r13                                              *|
|*           #define t2  r14                                              *|
|*           #define t3  r14                                              *|
|*           #define t4  r15                                              *|
|*           #define t5  r15                                              *|
|*           #define t6  r16                                              *|
|*                                                                        *|
|*           #define a0  f0                                               *|
|*           #define a1  f1                                               *|
|*           #define a2  f2                                               *|
|*           #define a3  f3                                               *|
|*           #define b0  f4                                               *|
|*           #define b1  f5                                               *|
|*           #define b2  f6                                               *|
|*           #define b3  f7                                               *|
|*           #define c0  f8                                               *|
|*           #define c1  f9                                               *|
|*           #define c2  f10                                              *|
|*           #define c3  f11                                              *|
|*           #define d0  f12                                              *|
|*           #define d1  f13                                              *|
|*           #define d2  f14                                              *|
|*           #define d3  f15                                              *|
|*                                                                        *|
|*        FUNC_PROLOG                     /* must precede function */     *|
|*                                                                        *|
|*        #if !defined( COMPILE_C )                                       *|
|*           U ENTRY(foo )                                                *|
|*           FORTRAN DREF 4(I, J, K, L)                                   *|
|*           FORTRAN_DREF_ARG8                                            *|
|*                                                                        *|
|*           U ENTRY(foo)                                                 *|
|*           LI(EFLAG, 0)                                                 *|
|*           BR(common)                                                   *|
|*                                                                        *|
|*           U ENTRY(foo x )                                              *|
|*           FORTRAN DREF 4(I, J, K, L)                                   *|
|*           FORTRAN DREF ARG8                                            *|
|*           FORTRAN_DREF_ARG9                                            *|
|*        #endif                                                          *|
```

salppc.h 2/23/2001

```
|*                                                                          *|
|*       ENTRY 10(foo x, A, I, B, J, C, K, D, L, N, EFLAG)                  *|
|*          DECLARE r13 r16                                                 *|
|*          DECLARE f0 f15                                                  *|
|*          GET_ARG9( EFLAG )   /* get the 9'th arg (EFLAG) off stack */   *|
|*                                                                          *|
|*       LABEL(common)                                                      *|
|*                                                                          *|
|*          SAVE CR             /* needed if using fields 2,3 or 4 */       *|
|*          SAVE r13 r16                                                    *|
|*          SAVE f14_f15                                                    *|
|*          SAVE_LR             /* needed if making a function call */      *|
|*                                                                          *|
|*          GET_ARG8( N )       /* get the 8'th arg (N) off stack */        *|
|*                                                                          *|
|*             /* ... body of function ... */                               *|
|*                                                                          *|
|*          REST CR                                                         *|
|*          REST r13 r16                                                    *|
|*          REST f14_f15                                                    *|
|*          REST LR                                                         *|
|*          RETURN                                                          *|
|*                                                                          *|
|*       FUNC_EPILOG                   /* must conclude function */         *|
|*                                                                          *|
|*           Mercury Computer Systems, Inc.                                 *|
|*           Copyright (c) 1996  All rights reserved                        *|
|*                                                                          *|
|* Revision      Date        Engineer; Reason                               *|
|* --------      ----        ----------------                               *|
|*   0.0        960223       jg;  Created                                   *|
|*   0.1        970109       jfk; Added POSTING BUFFER COUNT and made       *|
|*                                TEST IF DCBZ macro time "stw" instead     *|
|*                                of doing the TEST IF DCBT macro(lwz)      *|
|*   0.2        970124       jfk; Added SALCACHE ALLOC SIZE ,               *|
|*                                ALIGN SALCACHE, CREATE_SALCACHE_FRAME     *|
|*                                DESTROY SALCACHE FRAME                    *|
|*   0.3        970521       jfk; Added SET DCB[TZ] COND macros.            *|
|*                                Made old macros not assemble              *|
|*   0.4        980813       jfk; Changes SALCACHE ALLOC SIZE for 750       *|
+****************************************************************************+
#endif                                /* header */

#include <math.h>

#define uchar   unsigned char
#define ulong   unsigned long
#define ushort  unsigned short

#define CR   _cr
#define CTR  _ctr
#define VSCR _vscr

/*
 *  define a structure to represent a VMX register
 */
typedef union {
   char  c[16];
   uchar  uc[16];
   short  s[8];
   ushort  us[8];
   long  l[4];
   ulong  ul[4];
   float  f[4];
} VMX_reg;

#define FUNC_PROLOG
```

salppc.h 2/23/2001

```
#define FUNC_EPILOG \
   }

#define TEXT_SECTION( logb2_align )

#define DATA_SECTION( logb2_align )

#define RODATA_SECTION( logb2_align )

/*
 *  macro for C extern declarations
 */
#define EXTERN_DATA( symbol ) \
   extern long  symbol;

#define EXTERN_FUNC( func ) \
   extern void  func( void );

/*
 *  macro for a global declaration
 */
#define GLOBAL( symbol )

/*
 *  macro for a local declaration
 */
#define LOCAL( symbol )

/*
 *  macros for creating static arrays
 */
#define START_ARRAY( type, name ) \
type name##[] = {

#define START C ARRAY( name ) START ARRAY( char, name )
#define START UC ARRAY( name ) START ARRAY( uchar, name )
#define START S ARRAY( name ) START ARRAY( short, name )
#define START US ARRAY( name ) START ARRAY( ushort, name )
#define START L ARRAY( name ) START ARRAY( long, name )
#define START UL ARRAY( name ) START ARRAY( ulong, name )
#define START_F_ARRAY( name ) START_ARRAY( float, name )

#define END_ARRAY \
};

#define DATA( d1 ) \
d1,

#define DATA2( d1, d2 ) \
d1, d2,

#define DATA4( d1, d2, d3, d4 ) \
d1, d2, d3, d4,

#define DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
d1, d2, d3, d4, d5, d6, d7, d8,

#define C DATA( d1 )    DATA( d1 )
#define UC DATA( d1 )   DATA( d1 )
#define S DATA( d1 )    DATA( d1 )
#define US DATA( d1 )   DATA( d1 )
#define L DATA( d1 )    DATA( d1 )
#define UL DATA( d1 )   DATA( d1 )
#define F_DATA( d1 )    DATA( d1 )

#if defined( LITTLE_ENDIAN )
```

salppc.h 2/23/2001

```
#define D_DATA( d1, d2 )    DATA2( d2, d1 )
#else
#define D_DATA( d1, d2 )    DATA2( d1, d2 )
#endif

#define C_DATA2( d1, d2 )    DATA2( d1, d2 )
#define UC_DATA2( d1, d2 )   DATA2( d1, d2 )
#define S_DATA2( d1, d2 )    DATA2( d1, d2 )
#define US_DATA2( d1, d2 )   DATA2( d1, d2 )
#define L_DATA2( d1, d2 )    DATA2( d1, d2 )
#define UL_DATA2( d1, d2 )   DATA2( d1, d2 )
#define F_DATA2( d1, d2 )    DATA2( d1, d2 )

#define C_DATA4( d1, d2, d3, d4 )    DATA4( d1, d2, d3, d4 )
#define UC_DATA4( d1, d2, d3, d4 )   DATA4( d1, d2, d3, d4 )
#define S_DATA4( d1, d2, d3, d4 )    DATA4( d1, d2, d3, d4 )
#define US_DATA4( d1, d2, d3, d4 )   DATA4( d1, d2, d3, d4 )
#define L_DATA4( d1, d2, d3, d4 )    DATA4( d1, d2, d3, d4 )
#define UL_DATA4( d1, d2, d3, d4 )   DATA4( d1, d2, d3, d4 )
#define F_DATA4( d1, d2, d3, d4 )    DATA4( d1, d2, d3, d4 )

#define C_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define UC_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define S_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define US_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define L_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define UL_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )
#define F_DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( d1, d2, d3, d4, d5, d6, d7, d8 )

/*
 *  macros for creating vmx permute masks (128-bits)
 */
#if defined( LITTLE_ENDIAN )

#define L_PERMUTE_MUNGE( l ) ( (l) ^ 0x1c1c1c1c )
#define S_PERMUTE_MUNGE( s ) ( (s) ^ 0x1e1e )
#define C_PERMUTE_MUNGE( c ) ( (c) ^ 0x1f )

#define L_INDEX_MUNGE( x ) ( (x) ^ 0x3 )
#define S_INDEX_MUNGE( x ) ( (x) ^ 0x7 )
#define C_INDEX_MUNGE( x ) ( (x) ^ 0xf )

#else

#define L_PERMUTE_MUNGE( l ) ( l )
#define S_PERMUTE_MUNGE( s ) ( s )
#define C_PERMUTE_MUNGE( c ) ( c )

#define L_INDEX_MUNGE( x ) ( x )
#define S_INDEX_MUNGE( x ) ( x )
#define C_INDEX_MUNGE( x ) ( x )

#endif

#define L_PERMUTE_MASK( l1, l2, l3, l4 ) \
L_PERMUTE_MUNGE( l1 ), L_PERMUTE_MUNGE( l2 ), \
L_PERMUTE_MUNGE( l3 ), L_PERMUTE_MUNGE( l4 ),

#define S_PERMUTE_MASK( s1, s2, s3, s4, s5, s6, s7, s8 ) \
S_PERMUTE_MUNGE( s1 ), S_PERMUTE_MUNGE( s2 ), \
```

salppc.h 2/23/2001

```
S_PERMUTE_MUNGE( s3 ), S_PERMUTE_MUNGE( s4 ), \
S_PERMUTE_MUNGE( s5 ), S_PERMUTE_MUNGE( s6 ), \
S_PERMUTE_MUNGE( s7 ), S_PERMUTE_MUNGE( s8 ),

#define C_PERMUTE_MASK( c1, c2, c3, c4, c5, c6, c7, c8, \
                        c9, c10, c11, c12, c13, c14, c15, c16 ) \
C_PERMUTE_MUNGE( c1 ), C_PERMUTE_MUNGE( c2 ), \
C_PERMUTE_MUNGE( c3 ), C_PERMUTE_MUNGE( c4 ), \
C_PERMUTE_MUNGE( c5 ), C_PERMUTE_MUNGE( c6 ), \
C_PERMUTE_MUNGE( c7 ), C_PERMUTE_MUNGE( c8 ), \
C_PERMUTE_MUNGE( c9 ), C_PERMUTE_MUNGE( c10 ), \
C_PERMUTE_MUNGE( c11 ), C_PERMUTE_MUNGE( c12 ), \
C_PERMUTE_MUNGE( c13 ), C_PERMUTE_MUNGE( c14 ), \
C_PERMUTE_MUNGE( c15 ), C_PERMUTE_MUNGE( c16 ),

/*
 *  macro for a microcode entry point (e.g. vaddx, vaddx_)
 *  U_ENTRY is a "nop" for C code
 */
#define U_ENTRY( func_name )

/*
 *  macros for C function prototypes
 */
#define C_PROTOTYPE_0( func_name ) \
   void func_name ( void );

#define C_PROTOTYPE_1( func_name ) \
   void func_name ( long );

#define C_PROTOTYPE_2( func_name ) \
   void func_name ( long, long );

#define C_PROTOTYPE_3( func_name ) \
   void func_name ( long, long, long );

#define C_PROTOTYPE_4( func_name ) \
   void func_name ( long, long, long, long );

#define C_PROTOTYPE_5( func_name ) \
   void func_name ( long, long, long, long, long );

#define C_PROTOTYPE_6( func_name ) \
   void func_name ( long, long, long, long, long, long );

#define C_PROTOTYPE_7( func_name ) \
   void func_name ( long, long, long, long, long, long, long );

#define C_PROTOTYPE_8( func_name ) \
   void func_name ( long, long, long, long, long, long, long, long );

#define C_PROTOTYPE_9( func_name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long );

#define C_PROTOTYPE_10( func_name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long );

#define C_PROTOTYPE_11( func_name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long );

#define C_PROTOTYPE_12( func_name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long, long );
```

```
#define C PROTOTYPE_13( func name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long, long, long );

#define C PROTOTYPE_14( func name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long, long, long, long );

#define C PROTOTYPE_15( func name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long, long, long, long, long );

#define C PROTOTYPE_16( func name ) \
   void func_name ( long, long, long, long, long, long, long, long, \
                    long, long, long, long, long, long, long, long );

#define AUTO_r3 r31 \
   long  r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17,
   \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r4 r31 \
   long  r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r5 r31 \
   long  r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r6 r31 \
   long  r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r7 r31 \
   long  r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r8 r31 \
   long  r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO_r9 r31 \
   long  r9, r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO r10 r31 \
   long  r10, r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO r11 r31 \
   long  r11, r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO r12 r31 \
   long  r12, r13, r14, r15, r16, r17, \
         r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30,
         r31;
#define AUTO r13 r31 \
   long  r13, r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO r14 r31 \
   long  r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO r15 r31 \
   long  r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO_r16_r31 \
```

```
   long  r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO r17 r31 \
   long  r17, r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO r18 r31 \
   long  r18, r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;
#define AUTO r19 r31 \
   long  r19, r20, r21, r22, r23, r24, r25, \
         r26, r27, r28, r29, r30, r31;

#define AUTO f0 f31 \
   float  f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, \
          f15, f16, f17, f18, f19, f20, f21, f22, f23, f24, f25, f26, f27, \
          f28, f29, f30, f31;

#define AUTO d0 d31 \
   double  d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, \
           d15, d16, d17, d18, d19, d20, d21, d22, d23, d24, d25, d26, d27, \
           d28, d29, d30, d31;

#if defined( BUILD MAX )
#define AUTO v0_v31 \
   VMX_reg  v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14,
   \
          v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, \
          v28, v29, v30, v31;
#endif

/*
 *  For C implementation, create a dummy stack on function entry of size
 4096.
 */
#define STACK_SIZE  4096

/*
 *  macros for C and Fortran callable entry points
 */
#define ENTRY 0( func name ) \
   C PROTOTYPE 0( func name ) \
   void func_name ( void ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r3 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY 1( func name, arg0 ) \
   C PROTOTYPE 1( func name ) \
   void func_name ( long arg0 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r4 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;
```

salppc.h 2/23/2001

```
#define ENTRY_2( func_name, arg0, arg1 ) \
   C_PROTOTYPE_2( func_name ) \
   void func_name ( long arg0, long arg1 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r5 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
      long fpr_save_area[ 2*18 + 4 ]; \
      long vr_save_area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_3( func_name, arg0, arg1, arg2 ) \
   C_PROTOTYPE_3( func_name ) \
   void func_name ( long arg0, long arg1, long arg2 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r6 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
      long fpr_save_area[ 2*18 + 4 ]; \
      long vr_save_area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_4( func_name, arg0, arg1, arg2, arg3 ) \
   C_PROTOTYPE_4( func_name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r7 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
      long fpr_save_area[ 2*18 + 4 ]; \
      long vr_save_area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_5( func_name, arg0, arg1, arg2, arg3, arg4 ) \
   C_PROTOTYPE_5( func_name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r8 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
      long fpr_save_area[ 2*18 + 4 ]; \
      long vr_save_area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_6( func_name, arg0, arg1, arg2, arg3, arg4, arg5 ) \
   C_PROTOTYPE_6( func_name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r9 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
```

```
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_7( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6 ) \
   C PROTOTYPE 7( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r10 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_8( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6, arg7 ) \
   C PROTOTYPE 8( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r11 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_9( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6, arg7, arg8 ) \
   C PROTOTYPE 9( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r12 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_10( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9 ) \
   C PROTOTYPE 10( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r13 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;
```

```
#define ENTRY_11( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                              arg6, arg7, arg8, arg9, arg10 ) \
   C PROTOTYPE 11( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r14 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_12( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                              arg6, arg7, arg8, arg9, arg10, arg11 ) \
   C PROTOTYPE 12( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10, long arg11 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r15 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_13( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                              arg6, arg7, arg8, arg9, arg10, arg11, \
                              arg12 ) \
   C PROTOTYPE 13( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10, long arg11, long arg12 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r16 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_14( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                              arg6, arg7, arg8, arg9, arg10, arg11, \
                              arg12, arg13 ) \
   C PROTOTYPE 14( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10, long arg11, long arg12, long arg13 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r17 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr_save_area[ 19 + 4 ]; \
```

```
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_15( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12, arg13, arg14 ) \
   C PROTOTYPE 15( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10, long arg11, long arg12, long arg13, \
                    long arg14 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r18 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

#define ENTRY_16( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12, arg13, arg14, arg15 ) \
   C PROTOTYPE 16( func name ) \
   void func_name ( long arg0, long arg1, long arg2, long arg3, long arg4, \
                    long arg5, long arg6, long arg7, long arg8, long arg9, \
                    long arg10, long arg11, long arg12, long arg13, \
                    long arg14, long arg15 ) \
   { \
      long CR[8]; ulong CTR; ulong VSCR; long r0; \
      AUTO r19 r31 \
      AUTO f0 f31 \
      AUTO d0 d31 \
      AUTO_v0 v31 \
      long gpr save area[ 19 + 4 ]; \
      long fpr save area[ 2*18 + 4 ]; \
      long vr save area[ 4*12 + 4 ]; \
      long stack[STACK_SIZE + 4], sp;

/*
 *  macros to get GPR arguments beyond 8
 */
#define GET ARG8( rD )
#define GET ARG9( rD )
#define GET ARG10( rD )
#define GET ARG11( rD )
#define GET ARG12( rD )
#define GET ARG13( rD )
#define GET ARG14( rD )
#define GET ARG15( rD )
#define GET ARG16( rD )
#define GET_ARG17( rD )

/*
 *  macros to set GPR arguments beyond 8
 */
#define SET ARG8( rD )
#define SET ARG9( rD )
#define SET ARG10( rD )
#define SET ARG11( rD )
#define SET ARG12( rD )
#define SET ARG13( rD )
#define SET ARG14( rD )
#define SET_ARG15( rD )
```

salppc.h 2/23/2001

```
#define SET ARG16( rD )
#define SET_ARG17( rD )

/*
 *  macro to branch from one entry point to another
 */
#define BR FUNC( func_name ) \
   func_name (); \

/*
 *  macros to call functions
 */
#define CALL_FUNC( func_name ) \
   func_name ( );

/*
 *  macros to call functions
 */
#define CALL_0( func_name ) \
   func_name ( );

#define CALL_1( func name, arg0 ) \
   func_name ( arg0 );

#define CALL_2( func_name, arg0, arg1 ) \
   func_name ( arg0, arg1 );

#define CALL_3( func_name, arg0, arg1, arg2 ) \
   func_name ( arg0, arg1, arg2 );

#define CALL_4( func_name, arg0, arg1, arg2, arg3 ) \
   func_name ( arg0, arg1, arg2, arg3 );

#define CALL_5( func_name, arg0, arg1, arg2, arg3, arg4 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4 );

#define CALL_6( func_name, arg0, arg1, arg2, arg3, arg4, arg5 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5 );

#define CALL_7( func_name, arg0, arg1, arg2, arg3, arg4, arg5, arg6 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6 );

#define CALL_8( func_name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7 );

#define CALL_9( func_name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                arg8 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8 );

#define CALL_10( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9 );

#define CALL_11( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10 );

#define CALL_12( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10, arg11 );

#define CALL_13( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12 ) \
```

```
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10, arg11, arg12 );

#define CALL_14( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12, arg13 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10, arg11, arg12, arg13 );

#define CALL_15( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12, arg13, arg14 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10, arg11, arg12, arg13, arg14 );

#define CALL_16( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12, arg13, arg14, arg15 ) \
   func_name ( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
               arg8, arg9, arg10, arg11, arg12, arg13, arg14, arg15 );

#if defined( BUILD_MAX )

/*
 *  G4 macros to create a dummy jump table.
 *      (not supported in C)
 */
#define DECLARE VMX V1( root name )
#define DECLARE VMX V2( root name )
#define DECLARE VMX V3( root name )
#define DECLARE VMX V4( root name )
#define DECLARE_VMX_V5( root_name )

#define DECLARE VMX Z1( root name )
#define DECLARE VMX Z2( root name )
#define DECLARE VMX Z3( root name )
#define DECLARE VMX Z4( root name )
#define DECLARE_VMX_Z5( root_name )

/*
 *  G4 macros to decide whether to enter a VMX loop
 *  VMX loop is entered if at least minimum count,
 *  all vectors have the same relative alignment
 *  (i.e., same lower 4 bits) and all strides are unit.
 *  Note, a unit s imm argument is provided because some
 *  packed interleaved complex functions (stride 2) such
 *  as cvaddx() can be implemented with a VMX loop.
 *  Only one macro should be invoked per source file.
 *      (not supported in C)
 */
#define BR IF VMX V1( root name, min n imm, unit s imm, p1, s1, n, eflag )
#define BR_IF_VMX_V1_ALIGNED( root name, min n_imm, unit_s_imm, \
                              p1, s1, n, eflag )
#define BR_IF_VMX_V2( root name, min n imm, unit_s_imm, \
                      p1, s1, p2, s2, n, eflag )
#define BR_IF_VMX_V2_LS( root name, min n imm, unit_s_imm, \
                         p1, s1, ps, s2, n, eflag )
#define BR_IF_VMX_V2_LC( root name, min_n imm, unit_s_imm, \
                         p1, s1, pc, n, eflag )
#define BR_IF_VMX_V2_ALIGNED( root name, min n imm, unit_s_imm, \
                              p1, s1, p2, s2, n, eflag )
#define BR_IF_VMX_V3( root name, min n imm, unit_s imm, \
                      p1, s1, p2, s2, p3, s3, n, eflag )
#define BR_IF_VMX_V3_ALIGNED( root name, min n imm, unit_s imm, \
                              p1, s1, p2, s2, p3, s3, n, eflag )
#define BR_IF_VMX_V4( root name, min n imm, unit s imm, \
                      p1, s1, p2, s2, p3, s3, p4, s4, n, eflag )
#define BR_IF_VMX_V4_ALIGNED( root name, min n imm, unit s imm, \
                              p1, s1, p2, s2, p3, s3, p4, s4, n, eflag )
#define BR_IF_VMX_V5( root_name, min_n_imm, unit_s_imm, \
```

```
                      p1, s1, p2, s2, p3, s3, p4, s4, p5, s5, n, eflag )
#define BR_IF_VMX_V5_ALIGNED( root_name, min_n_imm, unit_s_imm, \
                              p1, s1, p2, s2, p3, s3, p4, s4, p5, s5, n,
                              eflag )
#define BR_IF_VMX_Z1( root_name, min_n_imm, unit_s_imm, \
                      pr1, pi1, s1, n, eflag )
#define BR_IF_VMX_Z2( root_name, min_n_imm, unit_s_imm, \
                      pr1, pi1, s1, pr2, pi2, s2, n, eflag )
#define BR_IF_VMX_Z3( root_name, min_n_imm, unit_s_imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, n, eflag )
#define BR_IF_VMX_Z4( root_name, min_n_imm, unit_s_imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, \
                      pr4, pi4, s4, n, eflag )
#define BR_IF_VMX_Z5( root_name, min_n_imm, unit_s_imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, \
                      pr4, pi4, s4, pr5, pi5, s5, n, eflag )
#define BR_IF_VMX_CONV( root_name, min_n_imm, \
                        p1, s1, s2, p3, s3, n, eflag )
#define BR_IF_VMX_ZCONV( root_name, min_n_imm, \
                         pr1, pi1, s1, s2, pr3, pi3, s3, n, eflag )

/*
 * G4 macro to get VMX unaligned (FP) count
 * assumes all vectors have the same relative alignment
 * and that the last 2 bits of ptr are 0
 * sets condition code CR0
 */
#define GET_VMX_UNALIGNED_COUNT( count, ptr ) \
   { \
      (count) = -(ptr); \
      (count) = ( (count) >> 2) & 3; \
      CR[0] = (long)(count); \
   }

/*
 * G4 macro to get VMX unaligned short count
 * assumes that the last bit of ptr is 0
 * sets condition code CR0
 */
#define GET_VMX_UNALIGNED_COUNT_S( count, ptr ) \
   { \
      (count) = -(ptr); \
      (count) = ( (count) >> 1) & 7; \
      CR[0] = (long)(count); \
   }

/*
 * G4 macro to get VMX unaligned char count
 * sets condition code CR0
 */
#define GET_VMX_UNALIGNED_COUNT_C( count, ptr ) \
   { \
      (count) = -(ptr); \
      (count) = (count) & 15; \
      CR[0] = (long)(count); \
   }

/*
 * G4 macro to load and splat an FP scalar independent of alignment
 */
#define SCALAR_SPLAT( vt, vtmp, scalarp ) \
   (vt).f[0] = (vt).f[1] = (vt).f[2] = (vt).f[3] = *scalarp;

#endif                              /* end BUILD_MAX */

/*
 *  cache (DCBT and DCBZ) macros.
```

salppc.h 2/23/2001

```
 */
#define DCBT TRUE( cond_bit, scratch ) \
   CR[(cond_bit)] = -1;              /* true (<= 0) */

#define DCBZ TRUE( cond_bit, scratch ) \
   DCBT_TRUE( cond_bit, scratch )

#define DCBT FALSE( cond_bit, scratch ) \
   CR[(cond_bit)] = 1;               /* false (> 0) */

#define DCBZ FALSE( cond_bit, scratch ) \
   DCBT_FALSE( cond_bit, scratch )

#define SET DCBT COND( cond bit, cache bit, eflag, scratch1 ) \
   CR[(cond_bit)] = (eflag & (cache_bit));

#define SET_DCBZ_COND( cond bit, cache bit, eflag, buffer, stride, \
                       unit stride, count, tmp1, tmp2, tmp3) \
   CR[(cond_bit)] = (eflag & (cache_bit));

#define DCBT IF( cond bit, rA, rB ) \
   if ( CR[(cond bit)] <= 0 ) \
      { DCBT( rA, rB ) }

#define DCBZ IF( cond bit, rA, rB ) \
   if ( CR[(cond bit)] <= 0 ) \
      { DCBZ( rA, rB ) }

#define DCBT IF CACHABLE( cond_bit, rA, rB ) \
   DCBT_IF( cond_bit, rA, rB )

#define DCBZ IF CACHABLE( cond_bit, rA, rB ) \
   DCBZ_IF( cond_bit, rA, rB )

#define BR IF CACHABLE( cond bit, label ) \
   if ( CR[(cond bit)] <= 0 ) \
      goto label;

#define BR IF NOT CACHABLE( cond_bit, label ) \
   if ( CR[(cond bit)] > 0 ) \
      goto label;

/*
 *  ASIC macros
 */
#if defined( COMPILE_PREFETCH )

#define LOAD PREFETCH CONTROL( mode, scratch1, scratch2 ) \
   *(volatile long *)PREFETCH_CONTROL = (mode);

#define LOAD MISCON B( mode, scratch1, scratch2 ) \
   *(volatile long *)MISCON_B = (mode);

#define RESET_PREFETCH_CONTROL( scratch1, scratch2 ) \
   { \
      volatile long  i; \
      i = *(volatile long *)MISCON_B; \
      i &= PREFETCH MASK; \
      i |= USE PREFETCH CONTROL; \
      *(volatile long *)PREFETCH_CONTROL = i; \
   }

#else

#define LOAD PREFETCH CONTROL( mode, scratch1, scratch2 )
#define LOAD MISCON B( mode, scratch1, scratch2 )
#define RESET_PREFETCH_CONTROL( scratch1, scratch2 )
```

```
#endif

/*
 *  instruction macros
 */
#define ADD( rD, rA, rB )                (rD) = (rA) + (rB);
#define ADD_C( rD, rA, rB )              (rD) = (rA) + (rB); CR[0] =
(long)(rD);
#define ADDI( rD, rA, SIMM )             (rD) = (rA) + (SIMM);
#define ADDIC_C( rD, rA, SIMM )          (rD) = (rA) + (SIMM); CR[0] = (long)(
rD);
#define ADDIS( rD, rA, SIMM )            (rD) = (rA) + ((SIMM) << 16);
#define AND( rA, rS, rB )                (rA) = (rS) & (rB);
#define AND_C( rA, rS, rB )              (rA) = (rS) & (rB); CR[0] =
(long)(rA);
#define ANDC( rA, rS, rB )               (rA) = (rS) & ~(rB);
#define ANDC_C( rA, rS, rB )             (rA) = (rS) & ~(rB); CR[0] =
(long)(rA);
#define ANDI_C( rA, rS, UIMM )           (rA) = (rS) & (UIMM); CR[0] = (long)(
rA);
#define ANDIS_C( rA, rS, UIMM )          (rA) = (rS) & ((UIMM) << 16); \
                                                CR[0] = (long)(rA);
#define BA( addr )                       goto (addr);
#define BCTR                             (*(void (*)(void))CTR)();
#define BEQ( label )                     if ( CR[0] == 0 ) goto label;
#define BEQ_PLUS( label )                BEQ( label )
#define BEQ_MINUS( label )               BEQ( label )
#define BEQ_CR( bit, label )             if ( CR[(bit)] == 0 ) goto label;
#define BEQ_CR_PLUS( bit, label )        BEQ_CR( bit, label )
#define BEQ_CR_MINUS( bit, label )       BEQ_CR( bit, label )
#define BEQLR                            if ( CR[0] == 0 ) return;
#define BEQLR_PLUS                       BEQLR
#define BEQLR_MINUS                      BEQLR
#define BEQLR_CR( bit )                  if ( CR[(bit)] == 0 ) return;
#define BEQLR_CR_PLUS( bit )             BEQLR_CR( bit )
#define BEQLR_CR_MINUS( bit )            BEQLR_CR( bit )
#define BGE( label )                     if ( CR[0] >= 0 ) goto label;
#define BGE_PLUS( label )                BGE( label )
#define BGE_MINUS( label )               BGE( label )
#define BGE_CR( bit, label )             if ( CR[(bit)] >= 0 ) goto label;
#define BGE_CR_PLUS( bit, label )        BGE_CR( bit, label )
#define BGE_CR_MINUS( bit, label )       BGE_CR( bit, label )
#define BGELR                            if ( CR[0] >= 0 ) return;
#define BGELR_PLUS                       BGELR
#define BGELR_MINUS                      BGELR
#define BGELR_CR( bit )                  if ( CR[(bit)] >= 0 ) return;
#define BGELR_CR_PLUS( bit )             BGELR_CR( bit )
#define BGELR_CR_MINUS( bit )            BGELR_CR( bit )
#define BGT( label )                     if ( CR[0] > 0 ) goto label;
#define BGT_PLUS( label )                BGT( label )
#define BGT_MINUS( label )               BGT( label )
#define BGT_CR( bit, label )             if ( CR[(bit)] > 0 ) goto label;
#define BGT_CR_PLUS( bit, label )        BGT_CR( bit, label )
#define BGT_CR_MINUS( bit, label )       BGT_CR( bit, label )
#define BGTLR                            if ( CR[0] > 0 ) return;
#define BGTLR_PLUS                       BGTLR
#define BGTLR_MINUS                      BGTLR
#define BGTLR_CR( bit )                  if ( CR[(bit)] > 0 ) return;
#define BGTLR_CR_PLUS( bit )             BGTLR_CR( bit )
#define BGTLR_CR_MINUS( bit )            BGTLR_CR( bit )
#define BL( func_name )                  func_name ( );
#define BLE( label )                     if ( CR[0] <= 0 ) goto label;
#define BLE_PLUS( label )                BLE( label )
#define BLE_MINUS( label )               BLE( label )
#define BLE_CR( bit, label )             if ( CR[(bit)] <= 0 ) goto label;
#define BLE_CR_PLUS( bit, label )        BLE_CR( bit, label )
```

```
#define BLE CR_MINUS( bit, label )       BLE CR( bit, label )
#define BLELR                            if ( CR[0] <= 0 ) return;
#define BLELR PLUS                       BLELR
#define BLELR MINUS                      BLELR
#define BLELR CR( bit )                  if ( CR[(bit)] <= 0 ) return;
#define BLELR CR PLUS( bit )             BLELR CR( bit )
#define BLELR_CR_MINUS( bit )            BLELR CR( bit )
#define BLR                              return;
#define BLT( label )                     if ( CR[0] < 0 ) goto label;
#define BLT PLUS( label )                BLT( label )
#define BLT MINUS( label )               BLT( label )
#define BLT CR( bit, label )             if ( CR[(bit)] < 0 ) goto label;
#define BLT CR PLUS( bit, label )        BLT CR( bit, label )
#define BLT CR_MINUS( bit, label )       BLT CR( bit, label )
#define BLTLR                            if ( CR[0] < 0 ) return;
#define BLTLR PLUS                       BLTLR
#define BLTLR MINUS                      BLTLR
#define BLTLR CR( bit )                  if ( CR[(bit)] < 0 ) return;
#define BLTLR CR PLUS( bit )             BLTLR CR( bit )
#define BLTLR CR MINUS( bit )            BLTLR CR( bit )
#define BNE( label )                     if ( CR[0] != 0 ) goto label;
#define BNE PLUS( label )                BNE( label )
#define BNE MINUS( label )               BNE( label )
#define BNE CR( bit, label )             if ( CR[(bit)] != 0 ) goto label;
#define BNE CR PLUS( bit, label )        BNE CR( bit, label )
#define BNE CR_MINUS( bit, label )       BNE CR( bit, label )
#define BNELR                            if ( CR[0] != 0 ) return;
#define BNELR PLUS                       BNELR
#define BNELR MINUS                      BNELR
#define BNELR CR( bit )                  if ( CR[(bit)] != 0 ) return;
#define BNELR CR PLUS( bit )             BNELR CR( bit )
#define BNELR CR MINUS( bit )            BNELR CR( bit )
#define BR( label )                      goto label;
#define CLRLWI( rA, rS, nbits )          (rA) = (rS) & ((1 << (32-nbits)) - 1);
#define CLRLWI_C( rA, rS, nbits )        (rA) = (rS) & ((1 << (32-nbits)) - 1);
\
                                                CR[0] = (long)(rA);
#define CLRRWI( rA, rS, nbits )          (rA) = (rS) & ~((1 << nbits) - 1);
#define CLRRWI_C( rA, rS, nbits )        (rA) = (rS) & ~((1 << nbits) - 1); \
                                                CR[0] = (long)(rA);
#define CMPLW( rA, rB )                  CR[0] = (((rA)^(rB)) & (1 << 31)) ? \
                                              ((rB) - (rA)) : ((rA) - (rB));
#define CMPLW_CR( bit, rA, rB )          CR[(bit)] = (((rA)^(rB)) & (1 << 31))
? \
                                              ((rB) - (rA)) : ((rA) - (rB));
#define CMPLWI( rA, UIMM )               CR[0] = (((rA)^(UIMM)) & (1 << 31)) ?
\
                                              ((UIMM) - (rA)) : ((rA) -
                                              (UIMM));
#define CMPLWI_CR( bit, rA, UIMM )       CR[(bit)] = (((rA)^(UIMM)) & (1 <<
31)) ? \
                                              ((UIMM) - (rA)) : ((rA) -
                                              (UIMM));
#define CMPW( rA, rB )                   CR[0] = (rA) - (rB);
#define CMPW CR( bit, rA, rB )           CR[(bit)] = (rA) - (rB);
#define CMPWI( rA, SIMM )                CR[0] = (rA) - (SIMM);
#define CMPWI_CR( bit, rA,. SIMM )       CR[(bit)] = (rA) - (SIMM);
#define DCBF( rA, rB )
#define DCBI( rA, rB )
#define DCBST( rA, rB )
#define DCBT( rA, rB )
#define DCBTST( rA, rB )
#define DCBZ( rA, rB )     *(long *)(((rA)+(rB)) & ~CACHE LINE MASK) = 0; \
                           *(long *)((((rA)+(rB)) & ~CACHE LINE MASK)+4) = 0; \
                           *(long *)((((rA)+(rB)) & ~CACHE LINE MASK)+8) = 0; \
                           *(long *)((((rA)+(rB)) & ~CACHE_LINE_MASK)+12) = 0;
                           \
```

```
                                  *(long *)((((rA)+(rB)) & ~CACHE_LINE_MASK)+16) = 0;
\
                                  *(long *)((((rA)+(rB)) & ~CACHE_LINE_MASK)+20) = 0;
\
                                  *(long *)((((rA)+(rB)) & ~CACHE_LINE_MASK)+24) = 0;
\
                                  *(long *)((((rA)+(rB)) & ~CACHE_LINE_MASK)+28) = 0;
#define DECR( rD )                          --(rD);
#define DECR C( rD )                        --(rD); CR[0] = (long)(rD);
#define DIVW( rD, rA, rB )                  (rD) = (rA) / (rB);
#define DIVW_C( rD, rA, rB )                (rD) = (rA) / (rB); CR[0] =
(long)(rD);
#define DIVWU( rD, rA, rB )                 (rD) = (ulong)(rA) / (ulong)(rB);
#define DIVWU_C( rD, rA, rB )               (rD) = (ulong)(rA) / (ulong)(rB); \
                                                CR[0] = (long)(rD);
#define EQV( rA, rS, rB )                   (rA) = ~((rS) ^ (rB));
#define EQV_C( rA, rS, rB )                 (rA) = ~((rS) ^ (rB)); \
                                            CR[0] = (long)(rA);
#define FABS( frD, frB )                    (frD) = ((frB) >= 0.0) ? (frB) :
-(frB);
#define FADD( frD, frA, frB )               (frD) = (frA) + (frB);
#define FADDS( frD, frA, frB )              (frD) = (frA) + (frB);
#define FCMPO( bit, frA, frB ) \
   { \
      if ( (frA) < (frB) ) CR[(bit)] = -1; \
      else if ( (frA) > (frB) ) CR[(bit)] = 1; \
      else CR[(bit)] = 0; \
   }
#define FCMPU( bit, frA, frB )              FCMPO( bit, frA, frB )
#define FCTIW( frD, frB )
#define FCTIWZ( frD, frB ) \
   { \
      union { \
         long i[2]; \
         double d; \
      } u; \
      u.i[0] = (long)(frB); \
      u.i[1] = 0; \
      (frD) = u.d; \
   }
#define FDIV( frD, frA, frB )               (frD) = (frA) / (frB);
#define FDIVS( frD, frA, frB )              (frD) = (frA) / (frB);
#define FMADD( frD, frA, frC, frB)          (frD) = (frA) * (frC) + (frB);
#define FMADDS( frD, frA, frC, frB)         (frD) = (frA) * (frC) + (frB);
#define FMOV( frD, frB )                    (frD) = (frB);
#define FMR( frD, frB )                     (frD) = (frB);
#define FMUL( frD, frA, frB )               (frD) = (frA) * (frB);
#define FMULS( frD, frA, frB )              (frD) = (frA) * (frB);
#define FMSUB( frD, frA, frC, frB )         (frD) = (frA) * (frC) - (frB);
#define FMSUBS( frD, frA, frC, frB )        (frD) = (frA) * (frC) - (frB);
#define FNABS( frD, frB )                   (frD) = ((frB) >= 0.0) ? -(frB) :
(frB);
#define FNEG( frD, frB )                    (frD) = -(frB);
#define FNMADD( frD, frA, frC, frB )        (frD) = -((frA) * (frC) + (frB));
#define FNMADDS( frD, frA, frC, frB )       (frD) = -((frA) * (frC) + (frB));
#define FNMSUB( frD, frA, frC, frB )        (frD) = -((frA) * (frC) - (frB));
#define FNMSUBS( frD, frA, frC, frB )       (frD) = -((frA) * (frC) - (frB));
#define FRES( frD, frB )
#define FRSP( frD, frB )                    (frD) = (float)(frB);
#define FRSQRTE( frD, frB )
#define FSEL( frD, frA, frC, frB )          (frD) = ((frA) >= 0.0) ? (frC) :
(frB);
#define FSUB( frD, frA, frB )               (frD) = (frA) - (frB);
#define FSUBS( frD, frA, frB )              (frD) = (frA) - (frB);
#define GOTO( label )                       BR( label )
#define INCR( rD )                          ++(rD);
#define INCR_C( rD )                        ++(rD); CR[0] = (long)(rD);
```

```
#define LA( rD, symbol, SIMM )              (rD) = (long)&(symbol);
#define LABEL( label )                      label:
#define LBZ( rD, rA, d )                    (rD) = *(uchar *)((rA) + (d));
#define LBZA( rD, symbol )                  (rD) = *(uchar *)&(symbol);
#define LBZU( rD, rA, d )                   (rD) = *(uchar *)((rA) += (d));
#define LBZUX( rD, rA, rB )                 (rD) = *(uchar *)((rA) += (rB));
#define LBZX( rD, rA, rB )                  (rD) = *(uchar *)((rA) + (rB));
#define LFD( frD, rA, d )                   (frD) = *(double *)((rA) + (d));
#define LFDU( frD, rA, d )                  (frD) = *(double *)((rA) += (d));
#define LFDUX( frD, rA, rB )                (frD) = *(double *)((rA) += (rB));
#define LFDX( frD, rA, rB )                 (frD) = *(double *)((rA) + (rB));
#define LFS( frD, rA, d )                   (frD) = *(float *)((rA) + (d));
#define LFSA( frD, symbol, rT )             (frD) = *(float *)&(symbol);
#define LFSU( frD, rA, d )                  (frD) = *(float *)((rA) += (d));
#define LFSUX( frD, rA, rB )                (frD) = *(float *)((rA) += (rB));
#define LFSX( frD, rA, rB )                 (frD) = *(float *)((rA) + (rB));
#define LHA( rD, rA, d )                    (rD) = *(short *)((rA) + (d));
#define LHAA( rD, symbol )                  (rD) = *(short *)&(symbol);
#define LHAU( rD, rA, d )                   (rD) = *(short *)((rA) += (d));
#define LHAUX( rD, rA, rB )                 (rD) = *(short *)((rA) += (rB));
#define LHAX( rD, rA, rB )                  (rD) = *(short *)((rA) + (rB));
#define LHZ( rD, rA, d )                    (rD) = *(ushort *)((rA) + (d));
#define LHZA( rD, symbol )                  (rD) = *(ushort *)&(symbol);
#define LHZU( rD, rA, d )                   (rD) = *(ushort *)((rA) += (d));
#define LHZUX( rD, rA, rB )                 (rD) = *(ushort *)((rA) += (rB));
#define LHZX( rD, rA, rB )                  (rD) = *(ushort *)((rA) + (rB));
#define LI( rD, SIMM )                      (rD) = (SIMM);
#define LIS( rD, SIMM )                     (rD) = ((SIMM) << 16);
#define LOAD_COUNT( rD )                    CTR = (rD);
#define LWZ( rD, rA, d )                    (rD) = *(long *)((rA) + (d));
#define LWZA( rD, symbol )                  (rD) = *(long *)&(symbol);
#define LWZU( rD, rA, d )                   (rD) = *(long *)((rA) += (d));
#define LWZUX( rD, rA, rB )                 (rD) = *(long *)((rA) += (rB));
#define LWZX( rD, rA, rB )                  (rD) = *(long *)((rA) + (rB));
#define MCRF( crfD, crfS )
#define MCRFS( crfD, crfS )
#define MFCR( rD )
#define MFCTR( rD )
#define MFLR( rD )
#define MFSPR( rD, SPR )
#define MOV( rA, rS )                       (rA) = (rS);
#define MOV_C( rA, rS )                     (rA) = (rS); CR[0] = (long)(rA);
#define MR( rA, rS )                        (rA) = (rS);
#define MR_C( rA, rS )                      (rA) = (rS); CR[0] = (long)(rA);
#define MTCR( rD )
#define MTCTR( rD )
#define MTFSFI( crfD, IMM )
#define MTLR( rD )
#define MTSPR( SPR, rS )
#define MULLI( rD, rA, SIMM )               (rD) = (rA) * (SIMM);
#define MULLW( rD, rA, rB )                 (rD) = (rA) * (rB);
#define MULLW_C( rD, rA, rB )               (rD) = (rA) * (rB); CR[0] =
(long)(rD);
#define NAND( rA, rS, rB )                  (rA) = ~((rS) & (rB));
#define NAND_C( rA, rS, rB )                (rA) = ~((rS) & (rB)); CR[0] = (long)(
rA);
#define NEG( rD, rA )                       (rD) = -(rA);
#define NEG_C( rD, rA )                     (rD) = -(rA); CR[0] = (long)(rA);
#define NOP
#define NOR( rA, rS, rB )                   (rA) = ~((rS) | (rB));
#define NOR_C( rA, rS, rB )                 (rA) = ~((rS) | (rB)); CR[0] = (long)(
rA);
#define OR( rA, rS, rB )                    (rA) = (rS) | (rB);
#define OR_C( rA, rS, rB )                  (rA) = (rS) | (rB); CR[0] =
(long)(rA);
#define ORC( rA, rS, rB )                   (rA) = (rS) | ~(rB);
#define ORC_C( rA, rS, rB )                 (rA) = (rS) | ~(rB); CR[0] =
```

```
(long)(rA);
#define ORI( rA, rS, UIMM )              (rA) = (rS) | (UIMM);
#define ORIS( rA, rS, UIMM )             (rA) = (rS) | ((UIMM) << 16);
#define RETURN                           BLR
#define RLWIMI( rA, rS, SH, MB, ME ) \
{ \
   ulong mask; \
   mask = ((1 << ((ME) - (MB) + 1)) - 1) << (31 - (ME)); \
   (rA) &= ~mask; \
   (rA) |= ((((rS) << (SH)) | ((ulong)(rS) >> (32 - (SH)))) & mask); \
}
#define RLWIMI_C( rA, rS, SH, MB, ME ) \
{ \
   ulong mask; \
   mask = ((1 << ((ME) - (MB) + 1)) - 1) << (31 - (ME)); \
   (rA) &= ~mask; \
   (rA) |= ((((rS) << (SH)) | ((ulong)(rS) >> (32 - (SH)))) & mask); \
   CR[0] = (long)(rA); \
}
#define RLWINM( rA, rS, SH, MB, ME ) \
{ \
   ulong mask; \
   mask = ((1 << ((ME) - (MB) + 1)) - 1) << (31 - (ME)); \
   (rA) = (((rS) << (SH)) | ((ulong)(rS) >> (32 - (SH)))) & mask; \
}
#define RLWINM_C( rA, rS, SH, MB, ME ) \
{ \
   ulong mask; \
   mask = ((1 << ((ME) - (MB) + 1)) - 1) << (31 - (ME)); \
   (rA) = (((rS) << (SH)) | ((ulong)(rS) >> (32 - (SH)))) & mask; \
   CR[0] = (long)(rA); \
}
#define RLWNM( rA, rS, rB, MB, ME )      RLWINM( rA, rS, (rB) & 0x1f, MB, ME )
#define RLWNM_C( rA, rS, rB, MB, ME )    RLWINM_C( rA, rS, (rB) & 0x1f, MB, ME
)
#define EXTLWI( rA, rS, n, b )           RLWINM( rA, rS, (b), 0, (n)-1 )
#define EXTLWI_C( rA, rS, n, b )         RLWINM_C( rA, rS, (b), 0, (n)-1 )
#define EXTRWI( rA, rS, n, b )           RLWINM( rA, rS, (b)+(n), 32-(n), 31 )
#define EXTRWI_C( rA, rS, n, b )         RLWINM( rA, rS, (b)+(n), 32-(n), 31 )
#define INSLWI( rA, rS, n, b )           RLWIMI( rA, rS, 32-(b), (b), (b)+(n)-1
)
#define INSLWI_C( rA, rS, n, b )         RLWIMI_C( rA, rS, 32-(b), (b),
(b)+(n)-1 )
#define INSRWI( rA, rS, n, b )           RLWIMI( rA, rS, 32-((b)+(n)), (b), (b)
+(n)-1 )
#define INSRWI_C( rA, rS, n, b )         RLWIMI_C( rA, rS, 32-((b)+(n)), (b), (
b)+(n)-1 )
#define ROTLW( rA, rS, rB )              RLWNM( rA, rS, rB, 0, 31 )
#define ROTLW_C( rA, rS, rB )            RLWNM_C( rA, rS, rB, 0, 31 )
#define ROTLWI( rA, rS, n )              RLWINM( rA, rS, (n), 0, 31 )
#define ROTLWI_C( rA, rS, n )            RLWINM_C( rA, rS, (n), 0, 31 )
#define ROTRWI( rA, rS, n )              RLWINM( rA, rS, 32-(n), 0, 31 )
#define ROTRWI_C( rA, rS, n )            RLWINM( rA, rS, 32-(n), 0, 31 )
#define SLW( rA, rS, rB )                (rA) = (rS) << (rB);
#define SLW_C( rA, rS, rB )              (rA) = (rS) << (rB); CR[0] =
(long)(rA);
#define SLWI( rA, rS, SH )               (rA) = (rS) << (SH);
#define SLWI_C( rA, rS, SH )             (rA) = (rS) << (SH); CR[0] =
(long)(rA);
#define SRAW( rA, rS, rB )               (rA) = (long)(rS) >> (rB);
#define SRAW_C( rA, rS, rB )             (rA) = (long)(rS) >> (rB); CR[0] = (
long)(rA);
#define SRAWI( rA, rS, SH )              (rA) = (long)(rS) >> (SH);
#define SRAWI_C( rA, rS, SH )            (rA) = (long)(rS) >> (SH); CR[0] = (
long)(rA);
#define SRW( rA, rS, rB )                (rA) = (ulong)(rS) >> (rB);
#define SRW_C( rA, rS, rB )              (rA) = (ulong)(rS) >> (rB); CR[0] = (
```

salppc.h 2/23/2001

```
long)(rA);
#define SRWI( rA, rS, SH )            (rA) = (ulong)(rS) >> (SH);
#define SRWI_C( rA, rS, SH )          (rA) = (ulong)(rS) >> (SH); CR[0] = (
long)(rA);
#define STB( rS, rA, d )              *(char *)((rA) + (d)) = (rS);
#define STBU( rS, rA, d )             *(char *)((rA) += (d)) = (rS);
#define STBUX( rS, rA, rB )           *(char *)((rA) += (rB)) = (rS);
#define STBX( rS, rA, rB )            *(char *)((rA) + (rB)) = (rS);
#define STFD( frD, rA, d )            *(double *)((rA) + (d)) = (frD);
#define STFDU( frD, rA, d )           *(double *)((rA) += (d)) = (frD);
#define STFDUX( frD, rA, rB )         *(double *)((rA) += (rB)) = (frD);
#define STFDX( frD, rA, rB )          *(double *)((rA) + (rB)) = (frD);
#define STFS( frD, rA, d )            *(float *)((rA) + (d)) = (frD);
#define STFSU( frD, rA, d )           *(float *)((rA) += (d)) = (frD);
#define STFSUX( frD, rA, rB )         *(float *)((rA) += (rB)) = (frD);
#define STFSX( frD, rA, rB )          *(float *)((rA) + (rB)) = (frD);
#define STH( rS, rA, d )              *(short *)((rA) + (d)) = (rS);
#define STHU( rS, rA, d )             *(short *)((rA) += (d)) = (rS);
#define STHUX( rS, rA, rB )           *(short *)((rA) += (rB)) = (rS);
#define STHX( rS, rA, rB )            *(short *)((rA) + (rB)) = (rS);
#define STW( rS, rA, d )              *(long *)((rA) + (d)) = (rS);
#define STWU( rS, rA, d )             *(long *)((rA) += (d)) = (rS);
#define STWUX( rS, rA, rB )           *(long *)((rA) += (rB)) = (rS);
#define STWX( rS, rA, rB )            *(long *)((rA) + (rB)) = (rS);
#define SUB( rD, rA, rB )             (rD) = (rA) - (rB);
#define SUB_C( rD, rA, rB )           (rD) = (rA) - (rB); CR[0] =
(long)(rD);
#define SUBFIC( rD, rA, SIMM )        (rD) = (SIMM) - (rA);
#define SUBI( rD, rA, SIMM )          (rD) = (rA) - (SIMM);
#define SUBIC_C( rD, rA, SIMM )       (rD) = (rA) - (SIMM); CR[0] = (long)(
rD);
#define SUBIS( rD, rA, SIMM )         (rD) = (rA) - ((SIMM) << 16);
#define TEST_COUNT( label )           if ( --CTR ) goto label;
#define XOR( rA, rS, rB )             (rA) = (rS) ^ (rB);
#define XOR_C( rA, rS, rB )           (rA) = (rS) ^ (rB); CR[0] =
(long)(rA);
#define XORI( rA, rS, UIMM )          (rA) = (rS) ^ (UIMM);
#define XORIS( rA, rS, UIMM )         (rA) = (rS) ^ ((UIMM) << 16);

#if defined( BUILD_MAX )

/*
 *  VMX instructions
 */
#define BR_VMX_ALL_TRUE( label )       if( CR[6] & 0x8 ) goto label;
#define BR_VMX_ALL_FALSE( label )      if( CR[6] & 0x2 ) goto label;
#define BR_VMX_NONE_TRUE( label )      if( CR[6] & 0x2 ) goto label;
#define BR_VMX_SOME_FALSE( label )     if( !(CR[6] & 0x8) ) goto label;
#define BR_VMX_SOME_TRUE( label )      if( !(CR[6] & 0x2) ) goto label;

#define DSS( STRM )
#define DSSALL
#define DST( rA, rB, STRM )
#define DSTT( rA, rB, STRM )
#define DSTST( rA, rB, STRM )
#define DSTSTT( rA, rB, STRM )

#if defined( COMPILE_NON_ALIGNED )
#define VMX_ADDR_MASK  0
#else
#define VMX_ADDR_MASK  15
#endif

#if defined( COMPILE_LVX_CHARS )

#define LVX( vT, rA, rB ) \
    { \
```

```
      char *addr; \
      ulong i; \
      addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
      for ( i = 0; i < 16; i++ ) \
         (vT).c[C_INDEX_MUNGE( i )] = addr[i]; \
   }
#define LVEBX( vT, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)((ulong)(rA) + (ulong)(rB)); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      (vT).c[C_INDEX_MUNGE( i )] = addr[0]; \
   }
#define LVEHX( vT, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      (vT).c[C_INDEX_MUNGE( i )] = addr[0]; \
      (vT).c[C_INDEX_MUNGE( i + 1 )] = addr[1]; \
   }
#define LVEWX( vT, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      (vT).c[C_INDEX_MUNGE( i )] = addr[0]; \
      (vT).c[C_INDEX_MUNGE( i + 1 )] = addr[1]; \
      (vT).c[C_INDEX_MUNGE( i + 2 )] = addr[2]; \
      (vT).c[C_INDEX_MUNGE( i + 3 )] = addr[3]; \
   }
#elif defined( COMPILE_LVX_SHORTS )

#define LVX( vT, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
      for ( i = 0; i < 8; i++ ) \
         (vT).s[S_INDEX_MUNGE( i )] = addr[i]; \
   }
#define LVEBX( vT, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)((ulong)(rA) + (ulong)(rB)); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      (vT).c[C_INDEX_MUNGE( i )] = addr[0]; \
   }
#define LVEHX( vT, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
      i = ((ulong)addr & VMX_ADDR_MASK) >> 1; \
      (vT).s[S_INDEX_MUNGE( i )] = addr[0]; \
   }
#define LVEWX( vT, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
      i = ((ulong)addr & VMX_ADDR_MASK) >> 1; \
      (vT).s[S_INDEX_MUNGE( i )] = addr[0]; \
```

```
        (vT).s[S_INDEX_MUNGE( i + 1 )] = addr[1]; \
    }

#else

#define LVX( vT, rA, rB ) \
    { \
        long *addr; \
        ulong i; \
        addr = (long *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
        for ( i = 0; i < 4; i++ ) \
            (vT).l[L_INDEX_MUNGE( i )] = addr[i]; \
    }
#define LVEBX( vT, rA, rB ) \
    { \
        char *addr; \
        ulong i; \
        addr = (char *)((ulong)(rA) + (ulong)(rB)); \
        i = (ulong)addr & VMX_ADDR_MASK; \
        (vT).c[C_INDEX_MUNGE( i )] = addr[0]; \
    }
#define LVEHX( vT, rA, rB ) \
    { \
        short *addr; \
        ulong i; \
        addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
        i = ((ulong)addr & VMX ADDR MASK) >> 1; \
        (vT).s[S_INDEX_MUNGE( i )] = addr[0]; \
    }
#define LVEWX( vT, rA, rB ) \
    { \
        long *addr; \
        ulong i; \
        addr = (long *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
        i = ((ulong)addr & VMX ADDR MASK) >> 2; \
        (vT).l[L_INDEX_MUNGE( i )] = addr[0]; \
    }

#endif

#if defined( COMPILE_STVX_CHARS )

#define STVX( vS, rA, rB ) \
    { \
        char *addr; \
        ulong i; \
        addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
        for ( i = 0; i < 16; i++ ) \
            addr[i] = (vS).c[C_INDEX_MUNGE( i )]; \
    }
#define STVEBX( vS, rA, rB ) \
    { \
        char *addr; \
        ulong i; \
        addr = (char *)((ulong)(rA) + (ulong)(rB)); \
        i = (ulong)addr & VMX ADDR MASK; \
        addr[0] = (vS).c[C_INDEX_MUNGE( i )]; \
    }
#define STVEHX( vS, rA, rB ) \
    { \
        char *addr; \
        ulong i; \
        addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
        i = (ulong)addr & VMX ADDR MASK; \
        addr[0] = (vS).c[C INDEX MUNGE( i )]; \
        addr[1] = (vS).c[C_INDEX_MUNGE( i + 1 )]; \
    }
```

```
#define STVEWX( vS, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      addr[0] = (vS).c[C_INDEX_MUNGE( i )]; \
      addr[1] = (vS).c[C_INDEX_MUNGE( i + 1 )]; \
      addr[2] = (vS).c[C_INDEX_MUNGE( i + 2 )]; \
      addr[3] = (vS).c[C_INDEX_MUNGE( i + 3 )]; \
   }

#elif defined( COMPILE_STVX_SHORTS )

#define STVX( vS, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
      for ( i = 0; i < 8; i++ ) \
         addr[i] = (vS).s[S_INDEX_MUNGE( i )]; \
   }
#define STVEBX( vS, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)((ulong)(rA) + (ulong)(rB)); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      addr[0] = (vS).c[C_INDEX_MUNGE( i )]; \
   }
#define STVEHX( vS, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
      i = ((ulong)addr & VMX_ADDR_MASK) >> 1; \
      addr[0] = (vS).s[S_INDEX_MUNGE( i )]; \
   }
#define STVEWX( vS, rA, rB ) \
   { \
      short *addr; \
      ulong i; \
      addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
      i = ((ulong)addr & VMX_ADDR_MASK) >> 1; \
      addr[0] = (vS).s[S_INDEX_MUNGE( i )]; \
      addr[1] = (vS).s[S_INDEX_MUNGE( i + 1 )]; \
   }

#else

#define STVX( vS, rA, rB ) \
   { \
      long *addr; \
      ulong i; \
      addr = (long *)(((ulong)(rA) + (ulong)(rB)) & ~VMX_ADDR_MASK); \
      for ( i = 0; i < 4; i++ ) \
         addr[i] = (vS).l[L_INDEX_MUNGE( i )]; \
   }
#define STVEBX( vS, rA, rB ) \
   { \
      char *addr; \
      ulong i; \
      addr = (char *)((ulong)(rA) + (ulong)(rB)); \
      i = (ulong)addr & VMX_ADDR_MASK; \
      addr[0] = (vS).c[C_INDEX_MUNGE( i )]; \
   }
#define STVEHX( vS, rA, rB ) \
```

```
    { \
        short *addr; \
        ulong i; \
        addr = (short *)(((ulong)(rA) + (ulong)(rB)) & ~1); \
        i = ((ulong)addr & VMX_ADDR_MASK) >> 1; \
        addr[0] = (vS).s[S_INDEX_MUNGE( i )]; \
    }
#define STVEWX( vS, rA, rB ) \
    { \
        long *addr; \
        ulong i; \
        addr = (long *)(((ulong)(rA) + (ulong)(rB)) & ~3); \
        i = ((ulong)addr & VMX_ADDR_MASK) >> 2; \
        addr[0] = (vS).l[L_INDEX_MUNGE( i )]; \
    }

#endif

#define LVSL_BE( vT, rA, rB ) \
    { \
        ulong i, j; \
        j = ((ulong)(rA) + (ulong)(rB)) & VMX_ADDR_MASK; \
        for ( i = 0; i < 16; i++ ) \
            (vT).uc[i] = j + i; \
    }
#define LVSR_BE( vT, rA, rB ) \
    { \
        ulong i, j; \
        j = 16 - (((ulong)(rA) + (ulong)(rB)) & VMX_ADDR_MASK); \
        for ( i = 0; i < 16; i++ ) \
            (vT).uc[i] = j + i; \
    }

#if defined( LITTLE_ENDIAN )
#define LVSL( vT, rA, rB )              LVSR_BE( vT, rA, rB );
#define LVSR( vT, rA, rB )              LVSL_BE( vT, rA, rB );
#else
#define LVSL( vT, rA, rB )              LVSL_BE( vT, rA, rB );
#define LVSR( vT, rA, rB )              LVSR_BE( vT, rA, rB );
#endif

#define LVXL( vT, rA, rB )              LVX( vT, rA, rB )
#define STVXL( vS, rA, rB )             STVX( vS, rA, rB )
#define VADDFP( vT, vA, vB ) \
    { \
        ulong i; \
        float a, b, c; \
        for ( i = 0; i < 4; i++ ) { \
            a = (vA).f[i]; \
            b = (vB).f[i]; \
            c = a + b; \
            (vT).f[i] = c; \
        } \
    }
#define VADDSBS( vT, vA, vB ) \
    { \
        ulong i; \
        long itemp; \
        for ( i = 0; i < 16; i++ ) { \
            itemp = (long)(vA).c[i] + (long)(vB).c[i]; \
            if ( itemp < -128 ) (vT).c[i] = -128; \
            else if ( itemp > 127 ) (vT).c[i] = 127; \
            else (vT).c[i] = (char)itemp; \
        } \
    }
#define VADDSHS( vT, vA, vB ) \
    { \
```

```
      ulong i; \
      long itemp; \
      for ( i = 0; i < 8; i++ ) { \
         itemp = (long)(vA).s[i] + (long)(vB).s[i]; \
         if ( itemp < -32768 ) (vT).s[i] = -32768; \
         else if ( itemp > 32767 ) (vT).s[i] = 32767; \
         else (vT).s[i] = (short)itemp; \
      } \
   }
#define VADDSWS( vT, vA, vB ) \
   { \
      ulong i; \
      long itemp; \
      for ( i = 0; i < 4; i++ ) { \
         itemp = (vA).l[i] + (vB).l[i]; \
         if ( ( (vA).l[i] > 0) && ( (vB).l[i] > 0) && (itemp < 0) ) \
            (vT).l[i] = (long)0x7fffffff; \
         else if ( ( (vA).l[i] < 0) && ( (vB).l[i] < 0) && (itemp > 0) ) \
            (vT).l[i] = (long)0x80000000; \
         else (vT).l = itemp[i]; \
      } \
   }
#define VADDUBM( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).uc[i] = (vA).uc[i] + (vB).uc[i]; \
   }
#define VADDUBS( vT, vA, vB ) \
   { \
      ulong i, itemp; \
      for ( i = 0; i < 16; i++ ) { \
         itemp = (ulong)(vA).uc[i] + (ulong)(vB).uc[i]; \
         if ( itemp > 255 ) (vT).uc[i] = 255; \
         else (vT).uc[i] = (uchar)itemp; \
      } \
   }
#define VADDUHM( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 8; i++ ) \
         (vT).us[i] = (vA).us[i] + (vB).us[i]; \
   }
#define VADDUHS( vT, vA, vB ) \
   { \
      ulong i, itemp; \
      for ( i = 0; i < 8; i++ ) { \
         itemp = (ulong)(vA).us[i] + (ulong)(vB).us[i]; \
         if ( itemp > 65535 ) (vT).uc[i] = 65535; \
         else (vT).uc[i] = (ushort)itemp; \
      } \
   }
#define VADDUWM( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = (vA).ul[i] + (vB).ul[i]; \
   }
#define VADDUWS( vT, vA, vB ) \
   { \
      ulong i, itemp; \
      for ( i = 0; i < 4; i++ ) { \
         itemp = (vA).ul[i] + (vB).ul[i]; \
         if ( itemp < (vA).ul[i] ) (vT).ul[i] = (ulong)0xffffffff; \
         else (vT).ul[i] = itemp; \
      } \
   }
```

salppc.h 2/23/2001

```
#define VAND( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = (vA).ul[i] & (vB).ul[i]; \
   }
#define VANDC( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = (vA).ul[i] & ~(vB).ul[i]; \
   }
#define VCMPEQFP( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = ( (vA).f[i] == (vB).f[i] ) ? 0xffffffff : 0; \
   }
#define VCMPEQFP_C( vT, vA, vB ) \
   { \
      ulong i; \
      ulong t, f; \
      t = 0xffffffff; \
      f = 0; \
      for ( i = 0; i < 4; i++ ) { \
         (vT).ul[i] = ( (vA).f[i] == (vB).f[i] ) ? 0xffffffff : 0; \
         t &= (vT).ul[i]; \
         f |= (vT).ul[i]; \
      } \
      if ( t ) CR[6] = 0x8; \
      else if ( !f ) CR[6] = 0x2; \
      else CR[6] = 0; \
   }
#define VCMPEQUB( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).uc[i] = ( (vA).uc[i] == (vB).uc[i] ) ? 0xff : 0; \
   }
#define VCMPEQUB_C( vT, vA, vB ) \
   { \
      ulong i; \
      uchar t, f; \
      t = 0xff; \
      f = 0; \
      for ( i = 0; i < 16; i++ ) { \
         (vT).uc[i] = ( (vA).uc[i] == (vB).uc[i] ) ? 0xff : 0; \
         t &= (vT).uc[i]; \
         f |= (vT).uc[i]; \
      } \
      if ( t ) CR[6] = 0x8; \
      else if ( !f ) CR[6] = 0x2; \
      else CR[6] = 0; \
   }
#define VCMPEQUH( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 8; i++ ) \
         (vT).us[i] = ( (vA).us[i] == (vB).us[i] ) ? 0xffff : 0; \
   }
#define VCMPEQUH_C( vT, vA, vB ) \
   { \
      ulong i; \
      ushort t, f; \
      t = 0xffff; \
      f = 0; \
      for ( i = 0; i < 8; i++ ) { \
```

```
            (vT).us[i] = ( (vA).us[i] == (vB).us[i] ) ? 0xffff : 0; \
            t &= (vT).us[i]; \
            f |= (vT).us[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPEQUW( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = ( (vA).ul[i] == (vB).ul[i] ) ? 0xffffffff : 0; \
    }
#define VCMPEQUW_C( vT, vA, vB ) \
    { \
        ulong i; \
        ulong t, f; \
        t = 0xffffffff; \
        f = 0; \
        for ( i = 0; i < 4; i++ ) { \
            (vT).ul[i] = ( (vA).ul[i] == (vB).ul[i] ) ? 0xffffffff : 0; \
            t &= (vT).ul[i]; \
            f |= (vT).ul[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPGEFP( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = ( (vA).f[i] >= (vB).f[i] ) ? 0xffffffff : 0; \
    }
#define VCMPGEFP_C( vT, vA, vB ) \
    { \
        ulong i; \
        ulong t, f; \
        t = 0xffffffff; \
        f = 0; \
        for ( i = 0; i < 4; i++ ) { \
            (vT).ul[i] = ( (vA).f[i] >= (vB).f[i] ) ? 0xffffffff : 0; \
            t &= (vT).ul[i]; \
            f |= (vT).ul[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPGTFP( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = ( (vA).f[i] > (vB).f[i] ) ? 0xffffffff : 0; \
    }
#define VCMPGTFP_C( vT, vA, vB ) \
    { \
        ulong i; \
        ulong t, f; \
        t = 0xffffffff; \
        f = 0; \
        for ( i = 0; i < 4; i++ ) { \
            (vT).ul[i] = ( (vA).f[i] > (vB).f[i] ) ? 0xffffffff : 0; \
            t &= (vT).ul[i]; \
            f |= (vT).ul[i]; \
        } \
```

```
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPGTSB( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 16; i++ ) \
            (vT).uc[i] = ( (vA).c[i] > (vB).c[i] ) ? 0xff : 0; \
    }
#define VCMPGTSB_C( vT, vA, vB ) \
    { \
        ulong i; \
        uchar t, f; \
        t = 0xff; \
        f = 0; \
        for ( i = 0; i < 16; i++ ) { \
            (vT).uc[i] = ( (vA).c[i] > (vB).c[i] ) ? 0xff : 0; \
            t &= (vT).uc[i]; \
            f |= (vT).uc[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPGTSH( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 8; i++ ) \
            (vT).us[i] = ( (vA).s[i] > (vB).s[i] ) ? 0xffff : 0; \
    }
#define VCMPGTSH_C( vT, vA, vB ) \
    { \
        ulong i; \
        ushort t, f; \
        t = 0xffff; \
        f = 0; \
        for ( i = 0; i < 8; i++ ) { \
            (vT).us[i] = ( (vA).s[i] > (vB).s[i] ) ? 0xffff : 0; \
            t &= (vT).us[i]; \
            f |= (vT).us[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
#define VCMPGTSW( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = ( (vA).l[i] > (vB).l[i] ) ? 0xffffffff : 0; \
    }
#define VCMPGTSW_C( vT, vA, vB ) \
    { \
        ulong i; \
        ulong t, f; \
        t = 0xffffffff; \
        f = 0; \
        for ( i = 0; i < 4; i++ ) { \
            (vT).ul[i] = ( (vA).l[i] > (vB).l[i] ) ? 0xffffffff : 0; \
            t &= (vT).ul[i]; \
            f |= (vT).ul[i]; \
        } \
        if ( t ) CR[6] = 0x8; \
        else if ( !f ) CR[6] = 0x2; \
        else CR[6] = 0; \
    }
```

saippc.h 2/23/2001

```
#define VCMPGTUB( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).uc[i] = ( (vA).uc[i] > (vB).uc[i] ) ? 0xff : 0; \
   }
#define VCMPGTUB_C( vT, vA, vB ) \
   { \
      ulong i; \
      uchar t, f; \
      t = 0xff; \
      f = 0; \
      for ( i = 0; i < 16; i++ ) { \
         (vT).uc[i] = ( (vA).uc[i] > (vB).uc[i] ) ? 0xff : 0; \
         t &= (vT).uc[i]; \
         f |= (vT).uc[i]; \
      } \
      if ( t ) CR[6] = 0x8; \
      else if ( !f ) CR[6] = 0x2; \
      else CR[6] = 0; \
   }
#define VCMPGTUH( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 8; i++ ) \
         (vT).us[i] = ( (vA).us[i] > (vB).us[i] ) ? 0xffff : 0; \
   }
#define VCMPGTUH_C( vT, vA, vB ) \
   { \
      ulong i; \
      ushort t, f; \
      t = 0xffff; \
      f = 0; \
      for ( i = 0; i < 8; i++ ) { \
         (vT).us[i] = ( (vA).us[i] > (vB).us[i] ) ? 0xffff : 0; \
         t &= (vT).us[i]; \
         f |= (vT).us[i]; \
      } \
      if ( t ) CR[6] = 0x8; \
      else if ( !f ) CR[6] = 0x2; \
      else CR[6] = 0; \
   }
#define VCMPGTUW( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = ( (vA).ul[i] > (vB).ul[i] ) ? 0xffffffff : 0; \
   }
#define VCMPGTUW_C( vT, vA, vB ) \
   { \
      ulong i; \
      ulong t, f; \
      t = 0xffffffff; \
      f = 0; \
      for ( i = 0; i < 4; i++ ) { \
         (vT).ul[i] = ( (vA).ul[i] > (vB).ul[i] ) ? 0xffffffff : 0; \
         t &= (vT).ul[i]; \
         f |= (vT).ul[i]; \
      } \
      if ( t ) CR[6] = 0x8; \
      else if ( !f ) CR[6] = 0x2; \
      else CR[6] = 0; \
   }
#define VCFSX( vT, vB, UIMM ) \
   { \
      float fj; \
      ulong i, j; \
```

```
      j = (127 - ((UIMM) & 0x1f)) << 23; \
      fj = *(float *)&j; \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = (float)((vB).l[i]) / fj; \
   }
#define VCFUX( vT, vB, UIMM ) \
   { \
      float fj; \
      ulong i, j; \
      j = (127 - ((UIMM) & 0x1f)) << 23; \
      fj = *(float *)&j; \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = (float)((vB).ul[i]) / fj; \
   }
#define VCTSXS( vT, vB, UIMM ) \
   { \
      float f, g, max, scale; \
      ulong i; \
      long l; \
      i = (127 + 31) << 23; \
      max = *(float *)&i; \
      i = (127 + ((UIMM) & 0x1f)) << 23; \
      scale = *(float *)&i; \
      for ( i = 0; i < 4; i++ ) { \
         f = (vB).f[i]; \
         g = f * scale; \
         if ( g <= -max ) l = 0x80000000; \
         else if ( g >= max ) l = 0x7fffffff; \
         else l = (long)f << ((UIMM) & 0x1f); \
         (vT).l[i] = l; \
      } \
   }
#define VCTUXS( vT, vB, UIMM ) \
   { \
      float f, g, max, scale; \
      ulong i, ul; \
      i = (127 + 32) << 23; \
      max = *(float *)&i; \
      i = (127 + ((UIMM) & 0x1f)) << 23; \
      scale = *(float *)&i; \
      for ( i = 0; i < 4; i++ ) { \
         f = (vB).f[i]; \
         g = f * scale; \
         if ( g <= 0 ) ul = 0; \
         else if ( g >= max ) ul = 0xffffffff; \
         else ul = (ulong)f << ((UIMM) & 0x1f); \
         (vT).ul[i] = ul; \
      } \
   }
#define VEXPTEFP( vT, vB ) \
   { \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = exp(0.693147180559945 * (vB).f[i]); \
   }
#define VLOGEFP( vT, vB ) \
   { \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = 1.442695040888963 * log((vB).f[i]); \
   }
#define VMADDFP( vT, vA, vC, vB ) \
   { \
      ulong i; \
      float a, b, c, d; \
      for ( i = 0; i < 4; i++ ) { \
         a = (vA).f[i]; \
         b = (vB).f[i]; \
         c = (vC).f[i]; \
```

```
        d = a * c; \
        d = b + d; \
        (vT).f[i] = d; \
     } \
  }
#define VMAXFP( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 4; i++ ) \
        (vT).f[i] = ((vA).f[i] >= (vB).f[i]) ? (vA).f[i] : (vB).f[i]; \
   }
#define VMAXSB( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 16; i++ ) \
        (vT).c[i] = ((vA).c[i] >= (vB).c[i]) ? (vA).c[i] : (vB).c[i]; \
   }
#define VMAXSH( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 8; i++ ) \
        (vT).s[i] = ((vA).s[i] >= (vB).s[i]) ? (vA).s[i] : (vB).s[i]; \
   }
#define VMAXSW( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 4; i++ ) \
        (vT).l[i] = ((vA).l[i] >= (vB).l[i]) ? (vA).l[i] : (vB).l[i]; \
   }
#define VMAXUB( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 16; i++ ) \
        (vT).uc[i] = ((vA).uc[i] >= (vB).uc[i]) ? (vA).uc[i] : (vB).uc[i]; \
   }
#define VMAXUH( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 8; i++ ) \
        (vT).us[i] = ((vA).us[i] >= (vB).us[i]) ? (vA).us[i] : (vB).us[i]; \
   }
#define VMAXUW( vT, vA, vB ) \
   { \
     ulong i; \
     for ( i = 0; i < 4; i++ ) \
        (vT).ul[i] = ((vA).ul[i] >= (vB).ul[i]) ? (vA).ul[i] : (vB).ul[i]; \
   }
#define VMHADDSHS( vD, vA, vB, vC ) \
   { \
     ulong i; \
     long a; \
     for ( i = 0; i < 8; i++ ) { \
        a = (long)(vA).s[i] * (long)(vB).s[i]; \
        a >>= 15; \
        a += (long)(vC).s[i]; \
        if ( a > 32767 ) a = 32767; \
        else if ( a < -32768 ) a = -32768; \
        (vD).s[i] = (short)a; \
     } \
   }
#define VMHRADDSHS( vD, vA, vB, vC ) \
   { \
     ulong i; \
     long a; \
     for ( i = 0; i < 8; i++ ) { \
        a = (long)(vA).s[i] * (long)(vB).s[i]; \
        a += 0x00004000; \
```

```
        a >>= 15; \
        a += (long)(vC).s[i]; \
        if ( a > 32767 ) a = 32767; \
        else if ( a < -32768 ) a = -32768; \
        (vD).s[i] = (short)a; \
      } \
   }
#define VMINFP( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = ((vA).f[i] <= (vB).f[i]) ? (vA).f[i] : (vB).f[i]; \
   }
#define VMINSB( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).c[i] = ((vA).c[i] <= (vB).c[i]) ? (vA).c[i] : (vB).c[i]; \
   }
#define VMINSH( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).s[i] = ((vA).s[i] <= (vB).s[i]) ? (vA).s[i] : (vB).s[i]; \
   }
#define VMINSW( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).l[i] = ((vA).l[i] <= (vB).l[i]) ? (vA).l[i] : (vB).l[i]; \
   }
#define VMINUB( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).uc[i] = ((vA).uc[i] <= (vB).uc[i]) ? (vA).uc[i] : (vB).uc[i]; \
   }
#define VMINUH( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).us[i] = ((vA).us[i] <= (vB).us[i]) ? (vA).us[i] : (vB).us[i]; \
   }
#define VMINUW( vT, vA, vB ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).ul[i] = ((vA).ul[i] <= (vB).ul[i]) ? (vA).ul[i] : (vB).ul[i]; \
   }
#define VMLADDUHM( vD, vA, vB, vC ) \
   { \
      ulong i; \
      ulong a, b, c; \
      for ( i = 0; i < 8; i++ ) { \
         a = (ulong)(vA).us[i]; \
         b = (ulong)(vB).us[i]; \
         c = (ulong)(vC).us[i]; \
         c += (a * b); \
         (vD).us[i] = (ushort)c; \
      } \
   }
#define VMR( vD, vS ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vD).ul[i] = (vS).ul[i]; \
   }
```

saippc.h 2/23/2001

```
#define VMRGHB_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 8; i++ ) { \
         j = i + i; \
         v.uc[j] = (vA).uc[i]; \
         v.uc[(j+1)] = (vB).uc[i]; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VMRGHH_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 4; i++ ) { \
         j = i + i; \
         v.us[j] = (vA).us[i]; \
         v.us[(j+1)] = (vB).us[i]; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VMRGHW_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 2; i++ ) { \
         j = i + i; \
         v.ul[j] = (vA).ul[i]; \
         v.ul[(j+1)] = (vB).ul[i]; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VMRGLB_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 8; i++ ) { \
         j = i + i; \
         v.uc[j] = (vA).uc[(8+i)]; \
         v.uc[(j+1)] = (vB).uc[(8+i)]; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VMRGLH_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 4; i++ ) { \
         j = i + i; \
         v.us[j] = (vA).us[(4+i)]; \
         v.us[(j+1)] = (vB).us[(4+i)]; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VMRGLW_BE( vT, vA, vB ) \
   { \
      VMX reg v; \
      ulong i, j; \
      for ( i = 0; i < 2; i++ ) { \
         j = i + i; \
         v.ul[j] = (vA).ul[(2+i)]; \
```

```
            v.ul[(j+1)] = (vB).ul[(2+i)]; \
        } \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = v.ul[i]; \
    }

#if defined( LITTLE_ENDIAN )
#define VMRGHB( vT, vA, vB )            VMRGLB BE( vT, vB, vA );
#define VMRGHH( vT, vA, vB )            VMRGLH BE( vT, vB, vA );
#define VMRGHW( vT, vA, vB )            VMRGLW BE( vT, vB, vA );
#define VMRGLB( vT, vA, vB )            VMRGHB BE( vT, vB, vA );
#define VMRGLH( vT, vA, vB )            VMRGHH BE( vT, vB, vA );
#define VMRGLW( vT, vA, vB )            VMRGHW_BE( vT, vB, vA );
#else
#define VMRGHB( vT, vA, vB )            VMRGHB BE( vT, vA, vB );
#define VMRGHH( vT, vA, vB )            VMRGHH BE( vT, vA, vB );
#define VMRGHW( vT, vA, vB )            VMRGHW BE( vT, vA, vB );
#define VMRGLB( vT, vA, vB )            VMRGLB BE( vT, vA, vB );
#define VMRGLH( vT, vA, vB )            VMRGLH BE( vT, vA, vB );
#define VMRGLW( vT, vA, vB )            VMRGLW_BE( vT, vA, vB );
#endif

#define VMSUMMBM( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        long a, c; \
        ulong b; \
        for ( i = 0; i < 4; i++ ) { \
            c = (vC).l[i]; \
            for ( j = 0; j < 4; j++ ) { \
                a = (long)(vA).c[4*i+j]; \
                b = (ulong)(vB).uc[4*i+j]; \
                c += (a * b); \
            } \
            (vT).l[i] = c; \
        } \
    }
#define VMSUMSHM( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        long a, b, c; \
        for ( i = 0; i < 4; i++ ) { \
            c = (vC).l[i]; \
            for ( j = 0; j < 2; j++ ) { \
                a = (long)(vA).s[4*i+j]; \
                b = (long)(vB).s[4*i+j]; \
                c += (a * b); \
            } \
            (vT).l[i] = c; \
        } \
    }
#define VMSUMSHS( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        long a, b; \
        double c; \
        for ( i = 0; i < 4; i++ ) { \
            c = (double)(vC).l[i]; \
            for ( j = 0; j < 2; j++ ) { \
                a = (long)(vA).s[4*i+j]; \
                b = (long)(vB).s[4*i+j]; \
                c += (double)(a * b); \
            } \
            if ( c >= 2147483647.0 ) c = 2147483647.0; \
            else if ( c <= -2147483648.0 ) c = -2147483648.0; \
            (vT).l[i] = (long)c; \
        } \
```

```
    }
#define VMSUMUBM( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        ulong a, b, c; \
        for ( i = 0; i < 4; i++ ) { \
            c = (vC).ul[i]; \
            for ( j = 0; j < 4; j++ ) { \
                a = (ulong)(vA).uc[4*i+j]; \
                b = (ulong)(vB).uc[4*i+j]; \
                c += (a * b); \
            } \
            (vT).ul[i] = c; \
        } \
    }
#define VMSUMUHM( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        ulong a, b, c; \
        for ( i = 0; i < 4; i++ ) { \
            c = (vC).ul[i]; \
            for ( j = 0; j < 2; j++ ) { \
                a = (ulong)(vA).us[4*i+j]; \
                b = (ulong)(vB).us[4*i+j]; \
                c += (a * b); \
            } \
            (vT).ul[i] = c; \
        } \
    }
#define VMSUMUHS( vT, vA, vB, vC ) \
    { \
        ulong i, j; \
        ulong a, b; \
        double c; \
        for ( i = 0; i < 4; i++ ) { \
            c = (double)(vC).ul[i]; \
            for ( j = 0; j < 2; j++ ) { \
                a = (ulong)(vA).us[4*i+j]; \
                b = (ulong)(vB).us[4*i+j]; \
                c += (double)(a * b); \
            } \
            if ( c >= 4294967295.0 ) c = 4294967295.0; \
            (vT).ul[i] = (ulong)c; \
        } \
    }
#define VMULESB( vT, vA, vB ) \
    { \
        ulong i; \
        long a, b, c; \
        for ( i = 0; i < 8; i++ ) { \
            a = (long)(vA).c[2*i]; \
            b = (long)(vB).c[2*i]; \
            c = a * b; \
            (vT).s[i] = (short)c; \
        } \
    }
#define VMULESH( vT, vA, vB ) \
    { \
        ulong i; \
        long a, b, c; \
        for ( i = 0; i < 4; i++ ) { \
            a = (long)(vA).s[2*i]; \
            b = (long)(vB).s[2*i]; \
            c = a * b; \
            (vT).l[i] = (long)c; \
        } \
    }
```

```
#define VMULEUB( vT, vA, vB ) \
   { \
      ulong i; \
      ulong a, b, c; \
      for ( i = 0; i < 8; i++ ) { \
         a = (ulong)(vA).uc[2*i]; \
         b = (ulong)(vB).uc[2*i]; \
         c = a * b; \
         (vT).us[i] = (ushort)c; \
      } \
   }
#define VMULEUH( vT, vA, vB ) \
   { \
      ulong i; \
      ulong a, b, c; \
      for ( i = 0; i < 4; i++ ) { \
         a = (ulong)(vA).us[2*i]; \
         b = (ulong)(vB).us[2*i]; \
         c = a * b; \
         (vT).ul[i] = (ulong)c; \
      } \
   }
#define VMULOSB( vT, vA, vB ) \
   { \
      ulong i; \
      long a, b, c; \
      for ( i = 0; i < 8; i++ ) { \
         a = (long)(vA).c[2*i+1]; \
         b = (long)(vB).c[2*i+1]; \
         c = a * b; \
         (vT).s[i] = (short)c; \
      } \
   }
#define VMULOSH( vT, vA, vB ) \
   { \
      ulong i; \
      long a, b, c; \
      for ( i = 0; i < 4; i++ ) { \
         a = (long)(vA).s[2*i+1]; \
         b = (long)(vB).s[2*i+1]; \
         c = a * b; \
         (vT).l[i] = (long)c; \
      } \
   }
#define VMULOUB( vT, vA, vB ) \
   { \
      ulong i; \
      ulong a, b, c; \
      for ( i = 0; i < 8; i++ ) { \
         a = (ulong)(vA).uc[2*i+1]; \
         b = (ulong)(vB).uc[2*i+1]; \
         c = a * b; \
         (vT).us[i] = (ushort)c; \
      } \
   }
#define VMULOUH( vT, vA, vB ) \
   { \
      ulong i; \
      ulong a, b, c; \
      for ( i = 0; i < 4; i++ ) { \
         a = (ulong)(vA).us[2*i+1]; \
         b = (ulong)(vB).us[2*i+1]; \
         c = a * b; \
         (vT).ul[i] = (ulong)c; \
      } \
   }
#define VNMSUBFP( vT, vA, vC, vB ) \
```

```
    { \
       ulong i; \
       float a, b, c, d; \
       for ( i = 0; i < 4; i++ ) { \
          a = (vA).f[i]; \
          b = (vB).f[i]; \
          c = (vC).f[i]; \
          d = a * c; \
          d = b - d; \
          (vT).f[i] = d; \
       } \
    }
#define VNOR( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = ~((vA).ul[i] | (vB).ul[i]); \
    }
#define VNOT( vT, vA )                    VNOR( vT, vA, vA )
#define VOR( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = (vA).ul[i] | (vB).ul[i]; \
    }
#define VPERM_BE( vT, vA, vB, vC ) \
    { \
       VMX_reg v; \
       ulong field, i; \
       for ( i = 0; i < 16; i++ ) { \
          field = (vC).uc[i]; \
          v.uc[i] = ( field < 16 ) ? (vA).uc[field] : (vB).uc[field - 16]; \
       } \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = v.ul[i]; \
    }
#define VPKUHUM_BE( vT, vA, vB, base ) \
    { \
       VMX_reg v; \
       ulong i, j; \
       j = base; \
       for ( i = 0; i < 8; i++ ) { \
          v.uc[i] = (vA).uc[(j)]; \
          v.uc[i+8] = (vB).uc[(j)]; \
          j += 2; \
       } \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = v.ul[i]; \
    }
#define VPKUHUS_BE( vT, vA, vB, base ) \
    { \
       VMX_reg v; \
       ulong i, j; \
       j = base; \
       for ( i = 0; i < 8; i++ ) { \
          v.uc[i] = (vA).uc[(j^1)] ? (uchar)255 : (vA).uc[(j)]; \
          v.uc[i+8] = (vB).uc[(j^1)] ? (uchar)255 : (vB).uc[(j)]; \
          j += 2; \
       } \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = v.ul[i]; \
    }
#define VPKSHUS_BE( vT, vA, vB, base ) \
    { \
       VMX_reg v; \
       ulong i, j; \
       j = base; \
```

```
      for ( i = 0; i < 8; i++ ) { \
         if ( (vA).s[i] <= 0 ) v.uc[i] = 0; \
         else if ( (vA).s[i] >= 255 ) v.uc[i] = 255; \
         else v.uc[i] = (vA).uc[j]; \
         if ( (vB).s[i] <= 0 ) v.uc[i+8] = 0; \
         else if ( (vB).s[i] >= 255 ) v.uc[i+8] = 255; \
         else v.uc[i+8] = (vB).uc[j]; \
         j += 2; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VPKSHSS_BE( vT, vA, vB, base ) \
   { \
      VMX reg v; \
      ulong i, j; \
      j = base; \
      for ( i = 0; i < 8; i++ ) { \
         if ( (vA).s[i] <= -128 ) v.c[i] = -128; \
         else if ( (vA).s[i] >= 127 ) v.c[i] = 127; \
         else v.c[i] = (vA).c[j]; \
         if ( (vB).s[i] <= -128 ) v.c[i+8] = -128; \
         else if ( (vB).s[i] >= 127 ) v.c[i+8] = 127; \
         else v.c[i+8] = (vB).c[j]; \
         j += 2; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VPKUWUM_BE( vT, vA, vB, base ) \
   { \
      VMX reg v; \
      ulong i, j; \
      j = base; \
      for ( i = 0; i < 4; i++ ) { \
         v.us[i] = (vA).us[(j)]; \
         v.us[i+4] = (vB).us[(j)]; \
         j += 2; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VPKUWUS_BE( vT, vA, vB, base ) \
   { \
      VMX reg v; \
      ulong i, j; \
      j = base; \
      for ( i = 0; i < 4; i++ ) { \
         v.us[i] = (vA).us[(j^1)] ? (ushort)65535 : (vA).us[(j)]; \
         v.us[i+4] = (vB).us[(j^1)] ? (ushort)65535 : (vB).us[(j)]; \
         j += 2; \
      } \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VPKSWUS_BE( vT, vA, vB, base ) \
   { \
      VMX reg v; \
      ulong i, j; \
      j = base; \
      for ( i = 0; i < 4; i++ ) { \
         if ( (vA).l[i] <= 0 ) v.us[i] = 0; \
         else if ( (vA).l[i] >= 65535 ) v.us[i] = 65535; \
         else v.us[i] = (vA).us[j]; \
         if ( (vB).l[i] <= 0 ) v.us[i+4] = 0; \
         else if ( (vB).l[i] >= 65535 ) v.us[i+4] = 65535; \
         else v.us[i+4] = (vB).us[j]; \
```

```
            j += 2; \
        } \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = v.ul[i]; \
    }
#define VPKSWSS_BE( vT, vA, vB, base ) \
    { \
        VMX_reg v; \
        ulong i, j; \
        j = base; \
        for ( i = 0; i < 8; i++ ) { \
            if ( (vA).l[i] <= -32768 ) v.s[i] = -32768; \
            else if ( (vA).l[i] >= 32767 ) v.s[i] = 32767; \
            else v.s[i] = (vA).s[j]; \
            if ( (vB).l[i] <= -32768 ) v.s[i+8] = -32768; \
            else if ( (vB).l[i] >= 32767 ) v.s[i+8] = 32767; \
            else v.s[i+8] = (vB).s[j]; \
            j += 2; \
        } \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = v.ul[i]; \
    }

#if defined( LITTLE_ENDIAN )
#define VPERM( vT, vA, vB, vC )         VPERM_BE( vT, vB, vA, vC );
#define VPKUHUM( vT, vA, vB )           VPKUHUM_BE( vT, vB, vA, 0 );
#define VPKUHUS( vT, vA, vB )           VPKUHUS_BE( vT, vB, vA, 0 );
#define VPKSHUS( vT, vA, vB )           VPKSHUS_BE( vT, vB, vA, 0 );
#define VPKSHSS( vT, vA, vB )           VPKSHSS_BE( vT, vB, vA, 0 );
#define VPKUWUM( vT, vA, vB )           VPKUWUM_BE( vT, vB, vA, 0 );
#define VPKUWUS( vT, vA, vB )           VPKUWUS_BE( vT, vB, vA, 0 );
#define VPKSWUS( vT, vA, vB )           VPKSWUS_BE( vT, vB, vA, 0 );
#define VPKSWSS( vT, vA, vB )           VPKSWSS_BE( vT, vB, vA, 0 );
#else
#define VPERM( vT, vA, vB, vC )         VPERM_BE( vT, vA, vB, vC );
#define VPKUHUM( vT, vA, vB )           VPKUHUM_BE( vT, vA, vB, 1 );
#define VPKUHUS( vT, vA, vB )           VPKUHUS_BE( vT, vA, vB, 1 );
#define VPKSHUS( vT, vA, vB )           VPKSHUS_BE( vT, vA, vB, 1 );
#define VPKSHSS( vT, vA, vB )           VPKSHSS_BE( vT, vA, vB, 1 );
#define VPKUWUM( vT, vA, vB )           VPKUWUM_BE( vT, vA, vB, 1 );
#define VPKUWUS( vT, vA, vB )           VPKUWUS_BE( vT, vA, vB, 1 );
#define VPKSWUS( vT, vA, vB )           VPKSWUS_BE( vT, vA, vB, 1 );
#define VPKSWSS( vT, vA, vB )           VPKSWSS_BE( vT, vA, vB, 1 );
#endif

#define VREFP( vT, vB ) \
    { \
        for ( i = 0; i < 4; i++ ) \
            (vT).f[i] = 1.0 / (vB).f[i]; \
    }
#define VRFIM( vT, vB ) \
    { \
        float f, max, r; \
        ulong i; \
        i = (127 + 31) << 23; \
        max = *(float *)&i; \
        for ( i = 0; i < 4; i++ ) { \
            f = (vB).f[i]; \
            if ( (f >= -max) && (f < max) ) { \
                r = (float)((long)f); \
                if ( r > f ) --r; \
                f = r; \
            } \
            (vT).f[i] = f; \
        } \
    }
#define VRFIN( vT, vB ) \
```

```
   { \
      float  f, r, s; \
      ulong i; \
      long lr; \
      for ( i = 0; i < 4; i++ ) { \
         s = f = (vB).f[i]; \
         if ( f < 0.0 ) f = -f; \
         r = f + 0.5; \
         if ( r != f ) { \
            lr = (long)r; \
            f = (float)lr; \
            if ( f == r ) f = (float)(lr & ~1); \
         } \
         if ( s < 0.0 ) f = -f; \
         (vT).f[i] = f; \
      } \
   }
#define VRFIP( vT, vB ) \
   { \
      float f, max, r; \
      ulong i; \
      i = (127 + 31) << 23; \
      max = *(float *)&i; \
      for ( i = 0; i < 4; i++ ) { \
         f = (vB).f[i]; \
         if ( (f >= -max) && (f < max) ) { \
            r = (float)((long)f); \
            if ( r < f ) ++r; \
            f = r; \
         } \
         (vT).f[i] = f; \
      } \
   }
#define VRFIZ( vT, vB ) \
   { \
      float f, max; \
      ulong i; \
      i = (127 + 31) << 23; \
      max = *(float *)&i; \
      for ( i = 0; i < 4; i++ ) { \
         f = (vB).f[i]; \
         if ( (f >= -max) && (f < max) ) \
            f = (float)((long)f); \
         (vT).f[i] = f; \
      } \
   }
#define VRLB( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 16; i++ ) { \
         sh = (vB).uc[i] & 0x7; \
         (vT).uc[i] = ((vA).uc[i] << sh) | ((vA).uc[i] >> (8-sh)); \
      } \
   }
#define VRLH( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 8; i++ ) { \
         sh = (vB).us[i] & 0xf; \
         (vT).us[i] = ((vA).us[i] << sh) | ((vA).us[i] >> (16-sh)); \
      } \
   }
#define VRSQRTEFP( vT, vB ) \
   { \
      for ( i = 0; i < 4; i++ ) \
         (vT).f[i] = 1.0 / sqrt((vB).f[i]); \
   }
```

```
#define VRLW( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 4; i++ ) { \
         sh = (vB).ul[i] & 0x1f; \
         (vT).ul[i] = ((vA).ul[i] << sh) | ((vA).ul[i] >> (32-sh)); \
      } \
   }
#define VSEL( vT, vA, vB, vC ) \
   { \
      ulong atemp, btemp, i; \
      for ( i = 0; i < 4; i++ ) { \
         atemp = (vA).ul[i] & ~(vC).ul[i]; \
         btemp = (vA).ul[i] & (vC).ul[i]; \
         (vT).ul[i] = atemp | btemp; \
      } \
   }
#define VSL( vT, vA, vB ) \
   { \
      ulong i, sh; \
      sh = (vB).ul[3] & 0x7; \
      (vT).ul[0] = ((vA).ul[0] << sh) | ((vA).ul[1] >> (32-sh)); \
      (vT).ul[1] = ((vA).ul[1] << sh) | ((vA).ul[2] >> (32-sh)); \
      (vT).ul[2] = ((vA).ul[2] << sh) | ((vA).ul[3] >> (32-sh)); \
      (vT).ul[3] = (vA).ul[3] << sh; \
   }
#define VSLDOI( vT, vA, vB, UIMM ) \
   { \
      VMX reg v; \
      ulong i, j, sh; \
      sh = (UIMM) & 0xf; \
      for ( i = 0; i < (16-sh); i++ ) \
         v.uc[i] = (vA).uc[i+sh]; \
      for ( j = i; j < 16; j++ ) \
         v.uc[j] = (vB).uc[j-i]; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = v.ul[i]; \
   }
#define VSLB( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 16; i++ ) { \
         sh = (vB).uc[i] & 0x7; \
         (vT).uc[i] = (vA).uc[i] << sh; \
      } \
   }
#define VSLH( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 8; i++ ) { \
         sh = (vB).us[i] & 0xf; \
         (vT).us[i] = (vA).us[i] << sh; \
      } \
   }
#define VSLO( vT, vA, vB ) \
   { \
      ulong i, j, sh; \
      sh = ((vB).ul[3] >> 3) & 0xf; \
      for ( i = 0; i < (16-sh); i++ ) \
         (vT).uc[i] = (vA).uc[i+sh]; \
      for ( j = i; j < 16; j++ ) \
         (vT).uc[j] = 0; \
   }
#define VSLW( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 4; i++ ) { \
```

```
         sh = (vB).ul[i] & 0x1f; \
         (vT).ul[i] = (vA).ul[i] << sh; \
      } \
   }
#define VSR( vT, vA, vB ) \
   { \
      ulong i, sh; \
      sh = (vB).ul[3] & 0x7; \
      (vT).ul[3] = ((vA).ul[3] >> sh) | ((vA).ul[2] << (32-sh)); \
      (vT).ul[2] = ((vA).ul[2] >> sh) | ((vA).ul[1] << (32-sh)); \
      (vT).ul[1] = ((vA).ul[1] >> sh) | ((vA).ul[0] << (32-sh)); \
      (vT).ul[0] = (vA).ul[0] >> sh; \
   }
#define VSRAB( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 16; i++ ) { \
         sh = (vB).uc[i] & 0x7; \
         (vT).c[i] = (vA).c[i] >> sh; \
      } \
   }
#define VSRAH( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 8; i++ ) { \
         sh = (vB).us[i] & 0xf; \
         (vT).s[i] = (vA).s[i] >> sh; \
      } \
   }
#define VSRAW( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 4; i++ ) { \
         sh = (vB).ul[i] & 0x1f; \
         (vT).l[i] = (vA).l[i] >> sh; \
      } \
   }
#define VSRB( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 16; i++ ) { \
         sh = (vB).uc[i] & 0x7; \
         (vT).uc[i] = (vA).uc[i] >> sh; \
      } \
   }
#define VSRH( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 8; i++ ) { \
         sh = (vB).us[i] & 0xf; \
         (vT).us[i] = (vA).us[i] >> sh; \
      } \
   }
#define VSRO( vT, vA, vB ) \
   { \
      long i, j, sh; \
      sh = ((vB).ul[3] >> 3) & 0xf; \
      for ( i = 15; i >= sh; i-- ) \
         (vT).uc[i] = (vA).uc[i-sh]; \
      for ( j = i; j >= 0; j-- ) \
         (vT).uc[j] = 0; \
   }
#define VSRW( vT, vA, vB ) \
   { \
      ulong i, sh; \
      for ( i = 0; i < 4; i++ ) { \
         sh = (vB).ul[i] & 0x1f; \
```

```
         (vT).ul[i] = (vA).ul[i] >> sh; \
      } \
   }
#define VSPLTB( vT, vB, UIMM ) \
   { \
      uchar c; \
      ulong i; \
      c = (vB).uc[C_INDEX_MUNGE( UIMM ) & 0xf]; \
      for ( i = 0; i < 16; i++ ) \
         (vT).uc[i] = c; \
   }
#define VSPLTH( vT, vB, UIMM ) \
   { \
      ushort s; \
      ulong i; \
      s = (vB).us[S_INDEX_MUNGE( UIMM ) & 0x7]; \
      for ( i = 0; i < 8; i++ ) \
         (vT).us[i] = s; \
   }
#define VSPLTW( vT, vB, UIMM ) \
   { \
      ulong i, l; \
      l = (vB).ul[L_INDEX_MUNGE( UIMM ) & 0x3]; \
      for ( i = 0; i < 4; i++ ) \
         (vT).ul[i] = l; \
   }
#define VSPLTISB( vT, SIMM ) \
   { \
      ulong i; \
      for ( i = 0; i < 16; i++ ) \
         (vT).c[i] = (char)(SIMM); \
   }
#define VSPLTISH( vT, SIMM ) \
   { \
      ulong i; \
      for ( i = 0; i < 8; i++ ) \
         (vT).s[i] = (short)(SIMM); \
   }
#define VSPLTISW( vT, SIMM ) \
   { \
      ulong i; \
      for ( i = 0; i < 4; i++ ) \
         (vT).l[i] = (long)(SIMM); \
   }
#define VSUBFP( vT, vA, vB ) \
   { \
      ulong i; \
      float a, b, c; \
      for ( i = 0; i < 4; i++ ) { \
         a = (vA).f[i]; \
         b = (vB).f[i]; \
         c = a - b; \
         (vT).f[i] = c; \
      } \
   }
#define VSUBSBS( vT, vA, vB ) \
   { \
      ulong i; \
      long itemp; \
      for ( i = 0; i < 16; i++ ) { \
         itemp = (long)(vA).c[i] - (long)(vB).c[i]; \
         if ( itemp < -128 ) (vT).c[i] = -128; \
         else if ( itemp > 127 ) (vT).c[i] = 127; \
         else (vT).c[i] = (char)itemp; \
      } \
   }
#define VSUBSHS( vT, vA, vB ) \
```

```
    { \
       ulong i; \
       long itemp; \
       for ( i = 0; i < 8; i++ ) { \
          itemp = (long)(vA).s[i] - (long)(vB).s[i]; \
          if ( itemp < -32768 ) (vT).s[i] = -32768; \
          else if ( itemp > 32767 ) (vT).s[i] = 32767; \
          else (vT).s[i] = (short)itemp; \
       } \
    }
#define VSUBSWS( vT, vA, vB ) \
    { \
       ulong i; \
       long itemp; \
       for ( i = 0; i < 4; i++ ) { \
          itemp = (vA).l[i] - (vB).l[i]; \
          if ( ( (vA).l[i] >= 0) && ( (vB).l[i] < 0) && (itemp < 0) ) \
             (vT).l[i] = (long)0x7fffffff; \
          else if ( ( (vA).l[i] < 0) && ( (vB).l[i] > 0) && (itemp > 0) ) \
             (vT).l[i] = (long)0x80000000; \
          else (vT).l = itemp[i]; \
       } \
    }
#define VSUBUBM( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 16; i++ ) \
          (vT).uc[i] = (vA).uc[i] - (vB).uc[i]; \
    }
#define VSUBUBS( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 16; i++ ) { \
          if ( (vA).uc[i] <= (vB).uc[i] ) (vT).uc[i] = 0; \
          else (vT).uc[i] = (vA).uc[i] - (vB).uc[i]; \
       } \
    }
#define VSUBUHM( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 8; i++ ) \
          (vT).us[i] = (vA).us[i] - (vB).us[i]; \
    }

#define VSUBUHS( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 8; i++ ) { \
          if ( (vA).us[i] <= (vB).us[i] ) (vT).us[i] = 0; \
          else (vT).us[i] = (vA).us[i] - (vB).us[i]; \
       } \
    }
#define VSUBUWM( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 4; i++ ) \
          (vT).ul[i] = (vA).ul[i] - (vB).ul[i]; \
    }
#define VSUBUWS( vT, vA, vB ) \
    { \
       ulong i; \
       for ( i = 0; i < 4; i++ ) { \
          if ( (vA).ul[i] <= (vB).ul[i] ) (vT).ul[i] = 0; \
          else (vT).ul[i] = (vA).ul[i] - (vB).ul[i]; \
       } \
    }
#define VSUMSWS( vT, vA, vB ) \
```

```
    { \
       ulong i; \
       double sum; \
       sum = (double)(vB).l[L_INDEX_MUNGE( 3 )]; \
       for ( i = 0; i < 4; i++ ) \
          sum += (double)(vA).l[i]; \
       if ( sum > (double)(0x7fffffff) ) \
          (vT).l[L_INDEX_MUNGE( 3 )] = 0x7fffffff; \
       else if ( sum < (double)(0x80000000) ) \
          (vT).l[L_INDEX_MUNGE( 3 )] = 0x80000000; \
       else \
          (vT).l[L_INDEX_MUNGE( 3 )] = (long)sum; \
    }
#define VSUM2SWS( vT, vA, vB ) \
    { \
       ulong i; \
       double sum1, sum2; \
       sum1 = (double)(vB).l[L_INDEX_MUNGE( 1 )]; \
       sum2 = (double)(vB).l[L_INDEX_MUNGE( 3 )]; \
       for ( i = 0; i < 2; i++ ) { \
          sum1 += (double)(vA).l[L_INDEX_MUNGE( i )]; \
          sum2 += (double)(vA).l[L_INDEX_MUNGE( i+2 )]; \
       } \
       if ( sum1 > (double)(0x7fffffff) ) \
          (vT).l[L_INDEX_MUNGE( 1 )] = 0x7fffffff; \
       else if ( sum1 < (double)(0x80000000) ) \
          (vT).l[L_INDEX_MUNGE( 1 )] = 0x80000000; \
       else \
          (vT).l[L_INDEX_MUNGE( 1 )] = (long)sum1; \
       if ( sum2 > (double)(0x7fffffff) ) \
          (vT).l[L_INDEX_MUNGE( 3 )] = 0x7fffffff; \
       else if ( sum2 < (double)(0x80000000) ) \
          (vT).l[L_INDEX_MUNGE( 3 )] = 0x80000000; \
       else \
          (vT).l[L_INDEX_MUNGE( 3 )] = (long)sum2; \
    }
#define VSUM4SBS( vT, vA, vB ) \
    { \
       ulong i, j; \
       double sum; \
       for ( i = 0; i < 4; i++ ) { \
          sum = (double)(vB).l[i]; \
          for ( j = 0; j < 4; j++ ) { \
             sum += (double)(vA).c[4*i + j]; \
             if ( sum > (double)(0x7fffffff) ) \
                (vT).l[i] = 0x7fffffff; \
             else if ( sum < (double)(0x80000000) ) \
                (vT).l[i] = 0x80000000; \
             else \
                (vT).l[i] = (long)sum; \
          } \
       } \
    }
#define VSUM4SHS( vT, vA, vB ) \
    { \
       ulong i, j; \
       double sum; \
       for ( i = 0; i < 4; i++ ) { \
          sum = (double)(vB).l[i]; \
          for ( j = 0; j < 2; j++ ) { \
             sum += (double)(vA).s[2*i + j]; \
             if ( sum > (double)(0x7fffffff) ) \
                (vT).l[i] = 0x7fffffff; \
             else if ( sum < (double)(0x80000000) ) \
                (vT).l[i] = 0x80000000; \
             else \
                (vT).l[i] = (long)sum; \
```

```
            } \
        } \
    }
#define VSUM4UBS( vT, vA, vB ) \
    { \
        ulong i, j; \
        double sum; \
        for ( i = 0; i < 4; i++ ) { \
            sum = (double)(vB).ul[i]; \
            for ( j = 0; j < 4; j++ ) { \
                sum += (double)(vA).uc[4*i + j]; \
                if ( sum > (2.0 * (double)(0x7fffffff) + 1.0) ) \
                    (vT).ul[i] = 0xffffffff; \
                else \
                    (vT).ul[i] = (ulong)sum; \
            } \
        } \
    }
#define VUPKHSB_BE( vT, vB ) \
    { \
        long i; \
        for ( i = 7; i >= 0; i-- ) \
            (vT).s[i] = (short)(vB).c[i]; \
    }
#define VUPKHSH_BE( vT, vB ) \
    { \
        long i; \
        for ( i = 3; i >= 0; i-- ) \
            (vT).l[i] = (long)(vB).s[i]; \
    }
#define VUPKLSB_BE( vT, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 8; i++ ) \
            (vT).s[i] = (short)(vB).c[i+8]; \
    }
#define VUPKLSH_BE( vT, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).l[i] = (long)(vB).s[i+4]; \
    }

#if defined( LITTLE_ENDIAN )
#define VUPKHSB( vT, vB )               VUPKLSB_BE( vT, vB );
#define VUPKHSH( vT, vB )               VUPKLSH_BE( vT, vB );
#define VUPKLSB( vT, vB )               VUPKHSB_BE( vT, vB );
#define VUPKLSH( vT, vB )               VUPKHSH_BE( vT, vB );
#else
#define VUPKHSB( vT, vB )               VUPKHSB_BE( vT, vB );
#define VUPKHSH( vT, vB )               VUPKHSH_BE( vT, vB );
#define VUPKLSB( vT, vB )               VUPKLSB_BE( vT, vB );
#define VUPKLSH( vT, vB )               VUPKLSH_BE( vT, vB );
#endif

#define VXOR( vT, vA, vB ) \
    { \
        ulong i; \
        for ( i = 0; i < 4; i++ ) \
            (vT).ul[i] = (vA).ul[i] ^ (vB).ul[i]; \
    }

#endif                                  /* end BUILD_MAX */

/*
 *  stack and register macros
 */
```

salppc.h 2/23/2001

```
#define VRSAVE_COND 7                    /* recommended VR condition bit */

/*
 *  macros to save and restore the CR register
 */
#define SAVE CR
#define REST_CR

/*
 *  macros to save and restore the LR register
 */
#define SAVE LR
#define REST_LR

/*
 *  GET FPR SAVE AREA places the start of the FPR save area into a register
 *  GET_GPR_SAVE_AREA places the start of the GPR save area into a register
 *
 *  For MAX only:
 *
 *     GET_VR_SAVE_AREA places the start of the VR save area into a register
 */
#define GET GPR SAVE AREA( ptr ) \
   ptr = (long)(((ulong)gpr_save_area + 15) & ~15);

#define GET FPR SAVE AREA( ptr ) \
   ptr = (long)(((ulong)fpr_save_area + 15) & ~15);

#if defined( BUILD MAX )
#define GET VR SAVE AREA( ptr ) \
   ptr = (long)(((ulong)vr_save_area + 15) & ~15);
#endif

/*
 *  macros to allocate and free space on the user stack.
 *  For C implementation, the size is limited to 4096 bytes.
 */
#define PUSH STACK( nbytes ) \
   sp = (long)(((ulong)stack + 15) & ~15);

#define POP_STACK( nbytes ) \
   sp = 0;

#define ALLOCATE STACK SPACE( ptr, nbytes ) \
   PUSH STACK( nbytes ) \
   ptr = sp;

#define FREE_STACK_SPACE( nbytes ) POP_STACK( nbytes )

#define CREATE_STACK FRAME( nbytes ) \
   PUSH_STACK( nbytes )

#define CREATE STACK FRAME X( nbytes ) \
   CREATE_STACK_FRAME( nbytes )

#define DESTROY_STACK_FRAME \
   sp = 0;

#define CREATE STACK BUFFER( bufferp, byte_align, nbytes ) \
   ALLOCATE_STACK_SPACE( bufferp, nbytes )

#define CREATE STACK BUFFER X( bufferp, byte_align, nbytes ) \
   CREATE_STACK_BUFFER( bufferp, byte_align, nbytes )

#define DESTROY_STACK_BUFFER \
   sp = 0;
```

salppc.h 2/23/2001

```
/*
 * macros to create salcache from the stack. used in ucode only
 */
#define CREATE STACK SALCACHE \
   char __localcachebuffer[SALCACHE_ALLOC_SIZE];

#define DESTROY_STACK_SALCACHE

/*
 *  macros for saving and restoring non-volatile
 *  floating point registers (FPRs)
 */
#define SAVE f14
#define SAVE f14 f15
#define SAVE f14 f16
#define SAVE f14 f17
#define SAVE f14 f18
#define SAVE f14 f19
#define SAVE f14 f20
#define SAVE f14 f21
#define SAVE f14 f22
#define SAVE f14 f23
#define SAVE f14 f24
#define SAVE f14 f25
#define SAVE f14 f26
#define SAVE f14 f27
#define SAVE f14 f28
#define SAVE f14 f29
#define SAVE f14 f30
#define SAVE_f14_f31

#define SAVE d14
#define SAVE d14 d15
#define SAVE d14 d16
#define SAVE d14 d17
#define SAVE d14 d18
#define SAVE d14 d19
#define SAVE d14 d20
#define SAVE d14 d21
#define SAVE d14 d22
#define SAVE d14 d23
#define SAVE d14 d24
#define SAVE d14 d25
#define SAVE d14 d26
#define SAVE d14 d27
#define SAVE d14 d28
#define SAVE d14 d29
#define SAVE d14 d30
#define SAVE_d14_d31

#define REST f14
#define REST f14 f15
#define REST f14 f16
#define REST f14 f17
#define REST f14 f18
#define REST f14 f19
#define REST f14 f20
#define REST f14 f21
#define REST f14 f22
#define REST f14 f23
#define REST f14 f24
#define REST f14 f25
#define REST f14 f26
#define REST f14 f27
#define REST f14 f28
#define REST f14 f29
#define REST_f14_f30
```

```
#define REST_f14_f31

#define REST d14
#define REST d14 d15
#define REST d14 d16
#define REST d14 d17
#define REST d14 d18
#define REST d14 d19
#define REST d14 d20
#define REST d14 d21
#define REST d14 d22
#define REST d14 d23
#define REST d14 d24
#define REST d14 d25
#define REST d14 d26
#define REST d14 d27
#define REST d14 d28
#define REST d14 d29
#define REST d14 d30
#define REST_d14_d31

/*
 *  macros for saving and restoring non-volatile
 *  general purpose registers (GPRs)
 */
#define SAVE r13
#define SAVE r13 r14
#define SAVE r13 r15
#define SAVE r13 r16
#define SAVE r13 r17
#define SAVE r13 r18
#define SAVE r13 r19
#define SAVE r13 r20
#define SAVE r13 r21
#define SAVE r13 r22
#define SAVE r13 r23
#define SAVE r13 r24
#define SAVE r13 r25
#define SAVE r13 r26
#define SAVE r13 r27
#define SAVE r13 r28
#define SAVE r13 r29
#define SAVE r13 r30
#define SAVE_r13_r31

#define REST r13
#define REST r13 r14
#define REST r13 r15
#define REST r13 r16
#define REST r13 r17
#define REST r13 r18
#define REST r13 r19
#define REST r13 r20
#define REST r13 r21
#define REST r13 r22
#define REST r13 r23
#define REST r13 r24
#define REST r13 r25
#define REST r13 r26
#define REST r13 r27
#define REST r13 r28
#define REST r13 r29
#define REST r13 r30
#define REST_r13_r31

#define SAVE r14
#define SAVE_r14_r15
```

salppc.h 2/23/2001

```
#define SAVE r14 r16
#define SAVE r14 r17
#define SAVE r14 r18
#define SAVE r14 r19
#define SAVE r14 r20
#define SAVE r14 r21
#define SAVE r14 r22
#define SAVE r14 r23
#define SAVE r14 r24
#define SAVE r14 r25
#define SAVE r14 r26
#define SAVE r14 r27
#define SAVE r14 r28
#define SAVE r14 r29
#define SAVE r14 r30
#define SAVE_r14_r31

#define REST r14
#define REST r14 r15
#define REST r14 r16
#define REST r14 r17
#define REST r14 r18
#define REST r14 r19
#define REST r14 r20
#define REST r14 r21
#define REST r14 r22
#define REST r14 r23
#define REST r14 r24
#define REST r14 r25
#define REST r14 r26
#define REST r14 r27
#define REST r14 r28
#define REST r14 r29
#define REST r14 r30
#define REST_r14_r31

#define SAVE r15
#define SAVE r15 r16
#define SAVE r15 r17
#define SAVE r15 r18
#define SAVE r15 r19
#define SAVE r15 r20
#define SAVE r15 r21
#define SAVE r15 r22
#define SAVE r15 r23
#define SAVE r15 r24
#define SAVE r15 r25
#define SAVE r15 r26
#define SAVE r15 r27
#define SAVE r15 r28
#define SAVE r15 r29
#define SAVE r15 r30
#define SAVE_r15_r31

#define REST r15
#define REST r15 r16
#define REST r15 r17
#define REST r15 r18
#define REST r15 r19
#define REST r15 r20
#define REST r15 r21
#define REST r15 r22
#define REST r15 r23
#define REST r15 r24
#define REST r15 r25
#define REST r15 r26
#define REST_r15_r27
```

salppc.h 2/23/2001

```
#define REST r15 r28
#define REST r15 r29
#define REST r15 r30
#define REST_r15_r31

#define SAVE r16
#define SAVE r16 r17
#define SAVE r16 r18
#define SAVE r16 r19
#define SAVE r16 r20
#define SAVE r16 r21
#define SAVE r16 r22
#define SAVE r16 r23
#define SAVE r16 r24
#define SAVE r16 r25
#define SAVE r16 r26
#define SAVE r16 r27
#define SAVE r16 r28
#define SAVE r16 r29
#define SAVE r16 r30
#define SAVE_r16_r31

#define REST r16
#define REST r16 r17
#define REST r16 r18
#define REST r16 r19
#define REST r16 r20
#define REST r16 r21
#define REST r16 r22
#define REST r16 r23
#define REST r16 r24
#define REST r16 r25
#define REST r16 r26
#define REST r16 r27
#define REST r16 r28
#define REST r16 r29
#define REST r16 r30
#define REST_r16_r31

/*
 *  VMX registers
 */
#define USE THRU v0( cond )
#define USE THRU v1( cond )
#define USE THRU v2( cond )
#define USE THRU v3( cond )
#define USE THRU v4( cond )
#define USE THRU v5( cond )
#define USE THRU v6( cond )
#define USE THRU v7( cond )
#define USE THRU v8( cond )
#define USE THRU v9( cond )
#define USE THRU v10( cond )
#define USE THRU v11( cond )
#define USE THRU v12( cond )
#define USE THRU v13( cond )
#define USE THRU v14( cond )
#define USE THRU v15( cond )
#define USE THRU v16( cond )
#define USE THRU v17( cond )
#define USE THRU v18( cond )
#define USE THRU v19( cond )
#define USE THRU v20( cond )
#define USE THRU v21( cond )
#define USE THRU v22( cond )
#define USE THRU v23( cond )
#define USE_THRU_v24( cond )
```

```
#define USE_THRU_v25( cond )
#define USE_THRU_v26( cond )
#define USE_THRU_v27( cond )
#define USE_THRU_v28( cond )
#define USE_THRU_v29( cond )
#define USE_THRU_v30( cond )
#define USE_THRU_v31( cond )

#define FREE_THRU_v0( cond )
#define FREE_THRU_v1( cond )
#define FREE_THRU_v2( cond )
#define FREE_THRU_v3( cond )
#define FREE_THRU_v4( cond )
#define FREE_THRU_v5( cond )
#define FREE_THRU_v6( cond )
#define FREE_THRU_v7( cond )
#define FREE_THRU_v8( cond )
#define FREE_THRU_v9( cond )
#define FREE_THRU_v10( cond )
#define FREE_THRU_v11( cond )
#define FREE_THRU_v12( cond )
#define FREE_THRU_v13( cond )
#define FREE_THRU_v14( cond )
#define FREE_THRU_v15( cond )
#define FREE_THRU_v16( cond )
#define FREE_THRU_v17( cond )
#define FREE_THRU_v18( cond )
#define FREE_THRU_v19( cond )
#define FREE_THRU_v20( cond )
#define FREE_THRU_v21( cond )
#define FREE_THRU_v22( cond )
#define FREE_THRU_v23( cond )
#define FREE_THRU_v24( cond )
#define FREE_THRU_v25( cond )
#define FREE_THRU_v26( cond )
#define FREE_THRU_v27( cond )
#define FREE_THRU_v28( cond )
#define FREE_THRU_v29( cond )
#define FREE_THRU_v30( cond )
#define FREE_THRU_v31( cond )

#endif                                  /* end SALPPC_H */

/*
 *--------------------------------------------------------------
 *
 *                    END OF FILE salppc.h
 *
 *--------------------------------------------------------------
 */
```

salppc.inc 3/9/2001

```
#if !defined( SALPPC_INC )
#define SALPPC_INC

#if 0
+****************************************************************************+
|***   MC Standard Algorithms -- PPC Version                              ***|
******************************************************************************
|*                                                                          *|
|*        File Name:      salppc.inc                                        *|
|*        Description:    SAL macro include file                            *|
|*                                                                          *|
|*        Source files should have extension .mac. For example, vadd.mac    *|
|*        and must include this file (salppc.inc).                          *|
|*                                                                          *|
|*        To assemble for PPC ucode, use the following basic                *|
|*        makefile build rule:                                              *|
|*                                                                          *|
|*           .SUFFIXES: .mac .c .s .o                                       *|
|*                                                                          *|
|*           .mac.o:                                                        *|
|*              cp $*.mac $*.c                                              *|
|*              ccmc -o $*.s -E $*.c                                        *|
|*              ccmc -c -o $*.o $*.s                                        *|
|*              rm -f $*.s                                                  *|
|*              rm -f $*.c                                                  *|
|*                                                                          *|
|*        To compile for C, use the following basic makefile build rule:    *|
|*                                                                          *|
|*           .SUFFIXES: .mac .c .o                                          *|
|*                                                                          *|
|*           .mac.o:                                                        *|
|*              cp $*.mac $*.c                                              *|
|*              ccmc -DCOMPILE_C -c -o $*.o $*.c                            *|
|*              rm -f $*.c                                                  *|
|*                                                                          *|
|*        The first 8 function arguments are passed in GPR registers        *|
|*        r3 - r10. Arguments beyond 8 are passed on the stack and may      *|
|*        be obtained with the GET_ARG8, GET_ARG9, ... GET_ARG15 macros.    *|
|*        Additional GPR registers should be assigned in ascending order    *|
|*        starting from the last function argument. These may be declared  *|
|*        with the DECLARE_rx[_ry] macros. For example, a function with     *|
|*        5 arguments that requires 3 additional GPR registers would        *|
|*        issue: DECLARE r8 r10. r0, if required, should be declared        *|
|*        separately with the DECLARE r0 macro. GPR registers above r12     *|
|*        must be saved and restored using the SAVE_r13[_ry] and            *|
|*        REST_r13[_ry] macros, respectively.                               *|
|*                                                                          *|
|*        FPR registers should be assigned in ascending order starting      *|
|*        with f0[d0]. These may be declared with the DECLARE_f0[_fy]       *|
|*        or DECLARE d0[_dy] macros.                                        *|
|*        For example, DECLARE f0 f11. FPR registers above f13[d13] must    *|
|*        be saved and restored using the SAVE f14[_fy] and REST f14[_fy]   *|
|*        or SAVE_d14[_dy] and REST_d14[_dy] macros, respectively.          *|
|*                                                                          *|
|*        All variables must be assigned a register using the               *|
|*        pre-processor #define directive. GPR registers are named          *|
|*        r0 - r31; Single precision FPR registers are named f0 - f31.      *|
|*        Double precision FPR registers are named d0 - d31. Different      *|
|*        variables may be assigned to the same register as in:             *|
|*                                                                          *|
|*           #define vara  f12                                              *|
|*           #define varb  f12                                              *|
|*                                                                          *|
|*        Functions must begin with the FUNC_PROLOG macro and end           *|
|*        with the FUNC_EPILOG macro.                                       *|
|*                                                                          *|
```

```
*        Macros are provided for both Fortran and C entry points.        *
*                                                                        *
*        The GET SALCACHE macro should be used to get the address of     *
*        the "current" salcache buffer into a GPR register.              *
*                                                                        *
*        Avoid terminating macro lines with a semicolon.                 *
*                                                                        *
*        The following example demonstrates typical usage:               *
*                                                                        *
*           #include "salppc.inc"                                        *
*                                                                        *
*           /*                                                           *
*            *  assign variables to registers                            *
*            */                                                          *
*           #define A  r3                                                *
*           #define I  r4                                                *
*           #define B  r5                                                *
*           #define J  r6                                                *
*           #define C  r7                                                *
*           #define K  r8                                                *
*           #define D  r9                                                *
*           #define L  r10                                               *
*           #define N  r12                                               *
*           #define EFLAG  r11                                           *
*           #define count  r11                                           *
*                                                                        *
*           #define t0  r13                                              *
*           #define t1  r13                                              *
*           #define t2  r14                                              *
*           #define t3  r14                                              *
*           #define t4  r15                                              *
*           #define t5  r15                                              *
*           #define t6  r16                                              *
*                                                                        *
*           #define a0  f0                                               *
*           #define a1  f1                                               *
*           #define a2  f2                                               *
*           #define a3  f3                                               *
*           #define b0  f4                                               *
*           #define b1  f5                                               *
*           #define b2  f6                                               *
*           #define b3  f7                                               *
*           #define c0  f8                                               *
*           #define c1  f9                                               *
*           #define c2  f10                                              *
*           #define c3  f11                                              *
*           #define d0  f12                                              *
*           #define d1  f13                                              *
*           #define d2  f14                                              *
*           #define d3  f15                                              *
*                                                                        *
*        FUNC_PROLOG                    /* must precede function */      *
*                                                                        *
*        #if !defined( COMPILE_C )                                       *
*           U ENTRY(foo )                                                *
*           FORTRAN DREF 4(I, J, K, L)                                   *
*           FORTRAN_DREF_ARG8                                            *
*                                                                        *
*           U ENTRY(foo)                                                 *
*           LI(EFLAG, 0)                                                 *
*           BR(common)                                                   *
*                                                                        *
*           U ENTRY(foo x )                                              *
*           FORTRAN DREF 4(I, J, K, L)                                   *
*           FORTRAN DREF ARG8                                            *
*           FORTRAN_DREF_ARG9                                            *
*        #endif                                                          *
```

```
|*                                                                          *|
|*        ENTRY 10(foo x, A, I, B, J, C, K, D, L, N, EFLAG)                 *|
|*           DECLARE r13 r16                                                *|
|*           DECLARE f0 f15                                                 *|
|*           GET_ARG9( EFLAG )    /* get the 9'th arg (EFLAG) off stack */  *|
|*                                                                          *|
|*        LABEL(common)                                                     *|
|*                                                                          *|
|*           SAVE CR                /* needed if using fields 2,3 or 4 */   *|
|*           SAVE r13 r16                                                   *|
|*           SAVE f14_f15                                                   *|
|*           SAVE_LR                /* needed if making a function call */  *|
|*                                                                          *|
|*           GET_ARG8( N )          /* get the 8'th arg (N) off stack */    *|
|*                                                                          *|
|*              /* ... body of function ... */                              *|
|*                                                                          *|
|*           REST CR                                                        *|
|*           REST r13 r16                                                   *|
|*           REST f14_f15                                                   *|
|*           REST LR                                                        *|
|*           RETURN                                                         *|
|*                                                                          *|
|*        FUNC_EPILOG                      /* must conclude function */     *|
|*                                                                          *|
|*            Mercury Computer Systems, Inc.                                *|
|*            Copyright (c) 1996  All rights reserved                       *|
|*                                                                          *|
|* Revision       Date        Engineer; Reason                              *|
|* --------       ----        ----------------                              *|
|*   0.0         960223       jg;  Created                                  *|
|*   0.1         970109       jfk; Added POSTING BUFFER COUNT and made      *|
|*                                 TEST IF DCBZ macro time "stw" instead    *|
|*                                 of doing the TEST IF DCBT macro(lwz)     *|
|*   0.2         970124       jfk; Added SALCACHE ALLOC SIZE ,              *|
|*                                 ALIGN SALCACHE, CREATE_SALCACHE_FRAME    *|
|*                                 DESTROY SALCACHE FRAME                   *|
|*   0.3         970521       jfk; Added SET DCB[TZ] COND macros.           *|
|*                                 Made old macros not assemble             *|
|*   0.4         980813       jfk; Changes SALCACHE ALLOC SIZE for 750      *|
+****************************************************************************+
#endif                                 /* header */

#if !defined( BUILD_603 ) && !defined( BUILD 750 ) && !defined( BUILD_MAX )
   #error  You must define BUILD_603 or BUILD_750 or BUILD_MAX
#endif

/*
 *  define single precision floating point field sizes,
 *  limits, and values
 */
#define F FLOAT SIZE  32
#define F FRAC SIZE  23
#define F HIDDEN SIZE  1
#define F EXP SIZE  8
#define F SIGN SIZE  1
#define F SIGN BIT  (F FLOAT SIZE - F SIGN SIZE)
#define F EXP MASK  ((1 << F EXP SIZE) - 1)
#define F EXP BIAS  ((1 << (F_EXP_SIZE-1)) - 1)
#define F MAX EXP  F EXP BIAS
#define F_MIN_EXP  (-(F_EXP_BIAS-1))

/*
 *  define double precision floating point field sizes,
 *  limits, and values
 */
#define D_FLOAT_SIZE  64
```

salppc.inc 3/9/2001

```
#define D FRAC SIZE  52
#define D HIDDEN SIZE  1
#define D EXP SIZE  11
#define D SIGN SIZE  1
#define D SIGN BIT  (D FLOAT SIZE - D SIGN SIZE)
#define D EXP MASK  ((1 << D EXP SIZE) - 1)
#define D EXP BIAS  ((1 << (D_EXP_SIZE-1)) - 1)
#define D MAX EXP  D EXP BIAS
#define D_MIN_EXP  (-(D_EXP_BIAS-1))

#if defined ( BUILD_603 )

#define  LOG2_CACHE_SIZE   (14)    /* Log (base 2) of 603 data cache */

#elif defined ( BUILD_750 ) || defined ( BUILD_MAX )

#define  LOG2_CACHE_SIZE   (15)    /* Log (base 2) of 750 or MAX data cache
*/

#endif

#define  LOG2 CACHE BSIZE  (LOG2 CACHE SIZE)
#define  LOG2 CACHE HSIZE  (LOG2 CACHE SIZE - 1)
#define  LOG2 CACHE LSIZE  (LOG2 CACHE SIZE - 2)
#define  LOG2 CACHE FSIZE  (LOG2 CACHE SIZE - 2)
#define  LOG2 CACHE DSIZE  (LOG2 CACHE SIZE - 3)
#define  LOG2 CACHE CSIZE  (LOG2 CACHE SIZE - 3)
#define  LOG2_CACHE_ZSIZE  (LOG2_CACHE_SIZE - 4)

#define  CACHE SIZE   (1 << LOG2 CACHE_SIZE)
#define  CACHE BSIZE  (CACHE SIZE)
#define  CACHE HSIZE  (CACHE SIZE >> 1)
#define  CACHE LSIZE  (CACHE SIZE >> 2)
#define  CACHE FSIZE  (CACHE SIZE >> 2)
#define  CACHE DSIZE  (CACHE SIZE >> 3)
#define  CACHE CSIZE  (CACHE SIZE >> 3)
#define  CACHE_ZSIZE  (CACHE_SIZE >> 4)

#define  LOG2 CACHE LINE_SIZE  5
#define  CACHE LINE SIZE  (1 << LOG2 CACHE_LINE SIZE)
#define  CACHE LINE LSIZE (CACHE LINE SIZE >> 2)
#define  CACHE LINE MASK  (CACHE LINE SIZE - 1)
#define  CACHE_LINE_ADDR_MASK  (0xffffffe0)

#define  LOG2 SALCACHE ALIGN  6
#define  SALCACHE ALIGN  (1 << LOG2 SALCACHE ALIGN)
#define  SALCACHE_ALIGN_MASK  (SALCACHE_ALIGN - 1)

#define  SALCACHE SIZE        CACHE SIZE
#define  SALCACHE EXTRA SIZE  (SALCACHE ALIGN + 64)
#define  SALCACHE_ALLOC_SIZE  (SALCACHE_SIZE + SALCACHE_EXTRA_SIZE)

/*
 *  Define memory vector non-cache (N) / cache (C) FLAG values for
 *  Enhanced SAL calls (final argument). The letters in the symbol
 *  correspond to the vectors in the call, moving from left to right
 *  so, for example:
 *
 *  for VMULX, there are the following 8 possibilities:
 *
 *     VMULX (A, I, B, J, C, K, N, SAL NNN)    A, B, C all not in cache
 *     VMULX (A, I, B, J, C, K, N, SAL NNC)    A, B not in cache, C in cache
 *     VMULX (A, I, B, J, C, K, N, SAL NCN)    A, C not in cache, B in cache
 *     VMULX (A, I, B, J, C, K, N, SAL NCC)    A not in cache, B, C in cache
 *     VMULX (A, I, B, J, C, K, N, SAL CNN)    B, C not in cache, A in cache
 *     VMULX (A, I, B, J, C, K, N, SAL CNC)    B not in cache, A, C in cache
 *     VMULX (A, I, B, J, C, K, N, SAL_CCN)    C not in cache, A, B in cache
```

```
 *     VMULX (A, I, B, J, C, K, N, SAL_CCC)    A, B, C all in cache
 */

/*
 *  1 vector algorithms
 */
#define  SAL_N  0
#define  SAL_C  1

/*
 *  2 vector algorithms
 */
#define  SAL_NN  0
#define  SAL_NC  1
#define  SAL_CN  2
#define  SAL_CC  3

/*
 *  3 vector algorithms
 */
#define  SAL_NNN  0
#define  SAL_NNC  1
#define  SAL_NCN  2
#define  SAL_NCC  3
#define  SAL_CNN  4
#define  SAL_CNC  5
#define  SAL_CCN  6
#define  SAL_CCC  7

/*
 *  4 vector algorithms
 */
#define  SAL_NNNN  0
#define  SAL_NNNC  1
#define  SAL_NNCN  2
#define  SAL_NNCC  3
#define  SAL_NCNN  4
#define  SAL_NCNC  5
#define  SAL_NCCN  6
#define  SAL_NCCC  7
#define  SAL_CNNN  8
#define  SAL_CNNC  9
#define  SAL_CNCN  10
#define  SAL_CNCC  11
#define  SAL_CCNN  12
#define  SAL_CCNC  13
#define  SAL_CCCN  14
#define  SAL_CCCC  15

/*
 *  5 vector algorithms
 */
#define  SAL_NNNNN  0
#define  SAL_NNNNC  1
#define  SAL_NNNCN  2
#define  SAL_NNNCC  3
#define  SAL_NNCNN  4
#define  SAL_NNCNC  5
#define  SAL_NNCCN  6
#define  SAL_NNCCC  7
#define  SAL_NCNNN  8
#define  SAL_NCNNC  9
#define  SAL_NCNCN  10
#define  SAL_NCNCC  11
#define  SAL_NCCNN  12
#define  SAL_NCCNC  13
#define  SAL_NCCCN  14
```

salppc.inc 3/9/2001

```
#define  SAL_NCCCC  15
#define  SAL_CNNNN  16
#define  SAL_CNNNC  17
#define  SAL_CNNCN  18
#define  SAL_CNNCC  19
#define  SAL_CNCNN  20
#define  SAL_CNCNC  21
#define  SAL_CNCCN  22
#define  SAL_CNCCC  23
#define  SAL_CCNNN  24
#define  SAL_CCNNC  25
#define  SAL_CCNCN  26
#define  SAL_CCNCC  27
#define  SAL_CCCNN  28
#define  SAL_CCCNC  29
#define  SAL_CCCCN  30
#define  SAL_CCCCC  31

/*
 *  define byte offsets into FFT_setup_ppc603e
 */
#define  FFT_SETUP_HANDLE              0
#define  FFT_SETUP_SMALL_TWIDP         4
#define  FFT_SETUP_SMALL_BITR_TWIDP    8
#define  FFT_SETUP_SMALL_LOG2M         12
#define  FFT_SETUP_BIG_TWIDP           16
#define  FFT_SETUP_BIG_XY_TWIDP        20
#define  FFT_SETUP_BIG_LOG2MXY         24
#define  FFT_SETUP_BIG_LOG2X           28
#define  FFT_SETUP_BIG_LOG2Y           32
#define  FFT_SETUP_BIG_STRIPX          36
#define  FFT_SETUP_RPASS_TWIDP         40
#define  FFT_SETUP_RADIX3_TWIDP        44
#define  FFT_SETUP_RADIX5_TWIDP        48
#define  FFT_SETUP_LOG2M               52
#define  FFT_SETUP_LOG2MR              56
#define  FFT_SETUP_VMX_BITR_TWIDP      60
#define  FFT_SETUP_VMX_TABLES          64

/*
 *  ASIC equates
 */
#define  ASIC_H                -1024          /* (0xFBFF + 1) */

#define  PREFETCH_CONTROL      (0xFBFFFE00)
#define  PREFETCH_CONTROL_H    -1024          /* (0xFBFF + 1) */
#define  PREFETCH_CONTROL_L    -512           /* (0xFE00) */

#define  MISCON_B              (0xFBFFFC18)
#define  MISCON_B_H            -1024          /* (0xFBFF + 1) */
#define  MISCON_B_L            -1000          /* (0xFC18) */

#define  PREFETCH_DISABLED     0
#define  PREFETCH_AUTO_6       1
#define  PREFETCH_AUTO_5       2
#define  PREFETCH_AUTO_4       3
#define  PREFETCH_AUTO_3       4
#define  PREFETCH_AUTO_2       5
#define  PREFETCH_AUTO_1       6
#define  PREFETCH_AUTO_0       7

#define  PREFETCH_MANUAL_0     8
#define  PREFETCH_MANUAL_2     9
#define  PREFETCH_MANUAL_4     10
#define  PREFETCH_MANUAL_6     11
#define  PREFETCH_MANUAL_8     12
#define  PREFETCH_MANUAL_10    13
```

salppc.inc 3/9/2001

```
#define  PREFETCH MANUAL 12  14
#define  PREFETCH_MANUAL_14  15

#define  USE PREFETCH_CONTROL  16
#define  USE_MISCON_B           0

#define  PREFETCH_MASK       15

#define  PREFETCH DEFAULT  (USE PREFETCH CONTROL | PREFETCH MANUAL 0)
#define  PREFETCH OFF      (USE PREFETCH CONTROL | PREFETCH DISABLED)
#define  PREFETCH A6       (USE PREFETCH CONTROL | PREFETCH AUTO 6)
#define  PREFETCH A5       (USE PREFETCH CONTROL | PREFETCH AUTO 5)
#define  PREFETCH A4       (USE PREFETCH CONTROL | PREFETCH AUTO 4)
#define  PREFETCH A3       (USE PREFETCH CONTROL | PREFETCH AUTO 3)
#define  PREFETCH A2       (USE PREFETCH CONTROL | PREFETCH AUTO 2)
#define  PREFETCH A1       (USE PREFETCH CONTROL | PREFETCH AUTO 1)
#define  PREFETCH_A0       (USE_PREFETCH_CONTROL | PREFETCH_AUTO_0)

#define  PREFETCH M0       (USE PREFETCH CONTROL | PREFETCH MANUAL 0)
#define  PREFETCH M2       (USE PREFETCH CONTROL | PREFETCH MANUAL 2)
#define  PREFETCH M4       (USE PREFETCH CONTROL | PREFETCH MANUAL 4)
#define  PREFETCH M6       (USE PREFETCH CONTROL | PREFETCH MANUAL 6)
#define  PREFETCH M8       (USE PREFETCH CONTROL | PREFETCH MANUAL 8)
#define  PREFETCH M10      (USE PREFETCH CONTROL | PREFETCH MANUAL 10)
#define  PREFETCH M12      (USE PREFETCH CONTROL | PREFETCH MANUAL 12)
#define  PREFETCH_M14      (USE_PREFETCH_CONTROL | PREFETCH_MANUAL_14)

/*
 *  macro to compile for PPC assembly (COMPILE_C *not* defined) or
 *  C code (COMPILE_C defined)
 */
#if defined( COMPILE_C )

#include "salppc.h"

#else

/*
 *  GPR register equates
 */
#define r0    0
#define sp    1
#define rtoc  2
#define r3    3
#define r4    4
#define r5    5
#define r6    6
#define r7    7
#define r8    8
#define r9    9
#define r10   10
#define r11   11
#define r12   12
#define r13   13
#define r14   14
#define r15   15
#define r16   16
#define r17   17
#define r18   18
#define r19   19
#define r20   20
#define r21   21
#define r22   22
#define r23   23
#define r24   24
#define r25   25
#define r26   26
```

```
#define r27   27
#define r28   28
#define r29   29
#define r30   30
#define r31   31

/*
 *  FPR single precision register equates
 */
#define f0    0
#define f1    1
#define f2    2
#define f3    3
#define f4    4
#define f5    5
#define f6    6
#define f7    7
#define f8    8
#define f9    9
#define f10   10
#define f11   11
#define f12   12
#define f13   13
#define f14   14
#define f15   15
#define f16   16
#define f17   17
#define f18   18
#define f19   19
#define f20   20
#define f21   21
#define f22   22
#define f23   23
#define f24   24
#define f25   25
#define f26   26
#define f27   27
#define f28   28
#define f29   29
#define f30   30
#define f31   31

/*
 *  FPR double precision register equates
 */
#define d0    0
#define d1    1
#define d2    2
#define d3    3
#define d4    4
#define d5    5
#define d6    6
#define d7    7
#define d8    8
#define d9    9
#define d10   10
#define d11   11
#define d12   12
#define d13   13
#define d14   14
#define d15   15
#define d16   16
#define d17   17
#define d18   18
#define d19   19
#define d20   20
#define d21   21
```

salppc.inc 3/9/2001

```
#define d22   22
#define d23   23
#define d24   24
#define d25   25
#define d26   26
#define d27   27
#define d28   28
#define d29   29
#define d30   30
#define d31   31

#if defined( BUILD_MAX )

/*
 *  VMX (g4) register equates
 */
#define v0    0
#define v1    1
#define v2    2
#define v3    3
#define v4    4
#define v5    5
#define v6    6
#define v7    7
#define v8    8
#define v9    9
#define v10   10
#define v11   11
#define v12   12
#define v13   13
#define v14   14
#define v15   15
#define v16   16
#define v17   17
#define v18   18
#define v19   19
#define v20   20
#define v21   21
#define v22   22
#define v23   23
#define v24   24
#define v25   25
#define v26   26
#define v27   27
#define v28   28
#define v29   29
#define v30   30
#define v31   31

#endif

#define FUNC PROLOG \
.section .text; \
.align 5;

#define FUNC_EPILOG

#define TEXT SECTION( logb2_align ) \
.section .text; \
.align logb2_align;

#define DATA SECTION( logb2_align ) \
.section .data; \
.align logb2_align;

#define RODATA SECTION( logb2_align ) \
.section .rodata; \
```

salppc.inc 3/9/2001

```
.align logb2_align;

#define PC_OFFSET( nbytes )  (. + (nbytes) )

/*
 *  make a "double" concat to fool the preprocessor so that input
 *  arguments get translated before concatenation; otherwise, the
 *  concatenated symbol doesn't get translated properly
 */
#define CONCAT( left, right ) CONCAT_NEST( left, right )
#define CONCAT_NEST( left, right ) left##right

/*
 *  macro for extern declarations and definitions
 */
#define EXTERN_DATA( symbol )

#define EXTERN_FUNC( func )

/*
 *  macro for a global declaration
 */
#define GLOBAL( symbol ) \
.globl symbol

/*
 *  macro for a local declaration
 */
#define LOCAL( symbol )

/*
 *  macros for creating static arrays
 */
#define START_ARRAY( name ) \
name##:

#define START_C_ARRAY( name ) START_ARRAY( name )
#define START_UC_ARRAY( name ) START_ARRAY( name )
#define START_S_ARRAY( name ) START_ARRAY( name )
#define START_US_ARRAY( name ) START_ARRAY( name )
#define START_L_ARRAY( name ) START_ARRAY( name )
#define START_UL_ARRAY( name ) START_ARRAY( name )
#define START_F_ARRAY( name ) START_ARRAY( name )

#define END_ARRAY

#define DATA( type, d1 ) \
.##type d1

#define DATA2( type, d1, d2 ) \
.##type d1, d2

#define DATA4( type, d1, d2, d3, d4 ) \
.##type d1, d2, d3, d4

#define DATA8( type, d1, d2, d3, d4, d5, d6, d7, d8 ) \
.##type d1, d2, d3, d4, d5, d6, d7, d8

#define C_DATA( d1 )   DATA( byte, d1 )
#define UC_DATA( d1 )  DATA( byte, d1 )
#define S_DATA( d1 )   DATA( short, d1 )
#define US_DATA( d1 )  DATA( short, d1 )
#define L_DATA( d1 )   DATA( long, d1 )
#define UL_DATA( d1 )  DATA( long, d1 )
#define F_DATA( d1 )   DATA( float, d1 )

#if defined( LITTLE_ENDIAN )
```

salppc.inc 3/9/2001

```
#define D_DATA( d1, d2 )   DATA2( long, d2, d1 )
#else
#define D_DATA( d1, d2 )   DATA2( long, d1, d2 )
#endif

#define C DATA2( d1, d2 )   DATA2( byte, d1, d2 )
#define UC DATA2( d1, d2 )  DATA2( byte, d1, d2 )
#define S DATA2( d1, d2 )   DATA2( short, d1, d2 )
#define US DATA2( d1, d2 )  DATA2( short, d1, d2 )
#define L DATA2( d1, d2 )   DATA2( long, d1, d2 )
#define UL DATA2( d1, d2 )  DATA2( long, d1, d2 )
#define F_DATA2( d1, d2 )   DATA2( float, d1, d2 )

#define C DATA4( d1, d2, d3, d4 )   DATA4( byte, d1, d2, d3, d4 )
#define UC DATA4( d1, d2, d3, d4 )  DATA4( byte, d1, d2, d3, d4 )
#define S DATA4( d1, d2, d3, d4 )   DATA4( short, d1, d2, d3, d4 )
#define US DATA4( d1, d2, d3, d4 )  DATA4( short, d1, d2, d3, d4 )
#define L DATA4( d1, d2, d3, d4 )   DATA4( long, d1, d2, d3, d4 )
#define UL DATA4( d1, d2, d3, d4 )  DATA4( long, d1, d2, d3, d4 )
#define F_DATA4( d1, d2, d3, d4 )   DATA4( float, d1, d2, d3, d4 )

#define C DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( byte, d1, d2, d3, d4, d5, d6, d7, d8 )
#define UC DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( byte, d1, d2, d3, d4, d5, d6, d7, d8 )
#define S DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( short, d1, d2, d3, d4, d5, d6, d7, d8 )
#define US DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( short, d1, d2, d3, d4, d5, d6, d7, d8 )
#define L DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( long, d1, d2, d3, d4, d5, d6, d7, d8 )
#define UL DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( long, d1, d2, d3, d4, d5, d6, d7, d8 )
#define F DATA8( d1, d2, d3, d4, d5, d6, d7, d8 ) \
        DATA8( float, d1, d2, d3, d4, d5, d6, d7, d8 )

/*
 *  macros for creating vmx permute masks (128-bits)
 */
#if defined( LITTLE_ENDIAN )

#define L PERMUTE MUNGE( l ) ( (l) ^ 0x1c1c1c1c )
#define S PERMUTE MUNGE( s ) ( (s) ^ 0x1e1e )
#define C_PERMUTE_MUNGE( c ) ( (c) ^ 0x1f )

#define L INDEX MUNGE( x ) ( (x) ^ 0x3 )
#define S INDEX MUNGE( x ) ( (x) ^ 0x7 )
#define C_INDEX_MUNGE( x ) ( (x) ^ 0xf )

#else

#define L PERMUTE MUNGE( l ) ( l )
#define S PERMUTE MUNGE( s ) ( s )
#define C_PERMUTE_MUNGE( c ) ( c )

#define L INDEX MUNGE( x ) ( x )
#define S INDEX MUNGE( x ) ( x )
#define C_INDEX_MUNGE( x ) ( x )

#endif

#define L PERMUTE MASK( l1, l2, l3, l4 ) \
.long L PERMUTE MUNGE( l1 ), L PERMUTE MUNGE( l2 ), \
      L_PERMUTE_MUNGE( l3 ), L_PERMUTE_MUNGE( l4 )

#define S PERMUTE MASK( s1, s2, s3, s4, s5, s6, s7, s8 ) \
.short S_PERMUTE_MUNGE( s1 ), S_PERMUTE_MUNGE( s2 ), \
```

```
        S PERMUTE MUNGE( s3 ), S PERMUTE MUNGE( s4 ), \
        S PERMUTE MUNGE( s5 ), S PERMUTE MUNGE( s6 ), \
        S_PERMUTE_MUNGE( s7 ), S_PERMUTE_MUNGE( s8 )

#define C_PERMUTE_MASK( c1, c2, c3, c4, c5, c6, c7, c8, \
                        c9, c10, c11, c12, c13, c14, c15, c16 ) \
.byte C PERMUTE MUNGE( c1 ), C PERMUTE MUNGE( c2 ), \
      C PERMUTE MUNGE( c3 ), C PERMUTE MUNGE( c4 ), \
      C PERMUTE MUNGE( c5 ), C PERMUTE MUNGE( c6 ), \
      C PERMUTE MUNGE( c7 ), C PERMUTE MUNGE( c8 ), \
      C PERMUTE MUNGE( c9 ), C PERMUTE MUNGE( c10 ), \
      C PERMUTE MUNGE( c11 ), C PERMUTE MUNGE( c12 ), \
      C PERMUTE MUNGE( c13 ), C PERMUTE MUNGE( c14 ), \
      C_PERMUTE_MUNGE( c15 ), C_PERMUTE_MUNGE( c16 )

/*
 *  macro for a microcode entry point (e.g. vaddx, vaddx_)
 *  U_ENTRY is a "nop" for C code
 */
#define U ENTRY( func_name ) \
.globl func_name; \
func_name:

/*
 *  macros for C function prototypes
 */
#define C PROTOTYPE 0( func name )
#define C PROTOTYPE 1( func name )
#define C PROTOTYPE 2( func name )
#define C PROTOTYPE 3( func name )
#define C PROTOTYPE 4( func name )
#define C PROTOTYPE 5( func name )
#define C PROTOTYPE 6( func name )
#define C PROTOTYPE 7( func name )
#define C PROTOTYPE 8( func name )
#define C PROTOTYPE 9( func name )
#define C PROTOTYPE 10( func name )
#define C PROTOTYPE 11( func name )
#define C PROTOTYPE 12( func name )
#define C PROTOTYPE 13( func name )
#define C PROTOTYPE 14( func name )
#define C PROTOTYPE 15( func name )
#define C_PROTOTYPE_16( func_name )

/*
 *  macros for C and Fortran callable entry points
 */
#define ENTRY 0( func_name ) \
.globl func_name; \
func_name:

#define ENTRY 1( func_name, arg0 ) \
.globl func_name; \
func_name:

#define ENTRY 2( func_name, arg0, arg1 ) \
.globl func_name; \
func_name:

#define ENTRY 3( func_name, arg0, arg1, arg2 ) \
.globl func_name; \
func_name:

#define ENTRY 4( func_name, arg0, arg1, arg2, arg3 ) \
.globl func_name; \
func_name:
```

salppc.inc 3/9/2001

```
#define ENTRY 5( func_name, arg0, arg1, arg2, arg3, arg4 ) \
.globl func_name; \
func_name:

#define ENTRY 6( func_name, arg0, arg1, arg2, arg3, arg4, arg5 ) \
.globl func_name; \
func_name:

#define ENTRY_7( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6 ) \
.globl func_name; \
func_name:

#define ENTRY_8( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6, arg7 ) \
.globl func_name; \
func_name:

#define ENTRY_9( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                            arg6, arg7, arg8 ) \
.globl func_name; \
func_name:

#define ENTRY_10( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9 ) \
.globl func_name; \
func_name:

#define ENTRY_11( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10 ) \
.globl func_name; \
func_name:

#define ENTRY_12( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11 ) \
.globl func_name; \
func_name:

#define ENTRY_13( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12 ) \
.globl func_name; \
func_name:

#define ENTRY_14( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12, arg13 ) \
.globl func_name; \
func_name:

#define ENTRY_15( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12, arg13, arg14 ) \
.globl func_name; \
func_name:

#define ENTRY_16( func_name, arg0, arg1, arg2, arg3, arg4, arg5, \
                             arg6, arg7, arg8, arg9, arg10, arg11, \
                             arg12, arg13, arg14, arg15 ) \
..globl func_name; \
func_name:

/*
 *  macros to de-reference any set of the first 8 arguments
 *  passed by reference to the Fortran entry point but by
 *  value to the corresponding C entry point
 */
```

salppc.inc 3/9/2001

```
#define FORTRAN DREF 1( arg0 ) \
   lwz arg0, 0(arg0);

#define FORTRAN DREF 2( arg0, arg1 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1);

#define FORTRAN DREF 3( arg0, arg1, arg2 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2);

#define FORTRAN DREF 4( arg0, arg1, arg2, arg3 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2); \
   lwz arg3, 0(arg3);

#define FORTRAN DREF 5( arg0, arg1, arg2, arg3, arg4 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2); \
   lwz arg3, 0(arg3); \
   lwz arg4, 0(arg4);

#define FORTRAN DREF 6( arg0, arg1, arg2, arg3, arg4, arg5 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2); \
   lwz arg3, 0(arg3); \
   lwz arg4, 0(arg4); \
   lwz arg5, 0(arg5);

#define FORTRAN DREF 7( arg0, arg1, arg2, arg3, arg4, arg5, arg6 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2); \
   lwz arg3, 0(arg3); \
   lwz arg4, 0(arg4); \
   lwz arg5, 0(arg5); \
   lwz arg6, 0(arg6);

#define FORTRAN DREF 8( arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7 ) \
   lwz arg0, 0(arg0); \
   lwz arg1, 0(arg1); \
   lwz arg2, 0(arg2); \
   lwz arg3, 0(arg3); \
   lwz arg4, 0(arg4); \
   lwz arg5, 0(arg5); \
   lwz arg6, 0(arg6); \
   lwz arg7, 0(arg7);

/*
 *  macros to de-reference specific arguments beyond the first 8
 *  passed by value to the C entry point
 */
#define ARG_OFF  (8 - 8*4)

#define FORTRAN DREF_ARG8 \
   lwz r12, (ARG_OFF + 8*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 8*4)(sp);

#define FORTRAN DREF_ARG9 \
   lwz r12, (ARG_OFF + 9*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 9*4)(sp);
```

```
#define FORTRAN_DREF_ARG10 \
   lwz r12, (ARG_OFF + 10*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 10*4)(sp);

#define FORTRAN_DREF_ARG11 \
   lwz r12, (ARG_OFF + 11*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 11*4)(sp);

#define FORTRAN_DREF_ARG12 \
   lwz r12, (ARG_OFF + 12*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 12*4)(sp);

#define FORTRAN_DREF_ARG13 \
   lwz r12, (ARG_OFF + 13*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 13*4)(sp);

#define FORTRAN_DREF_ARG14 \
   lwz r12, (ARG_OFF + 14*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 14*4)(sp);

#define FORTRAN_DREF_ARG15 \
   lwz r12, (ARG_OFF + 15*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 15*4)(sp);

#define FORTRAN_DREF_ARG16 \
   lwz r12, (ARG_OFF + 16*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 16*4)(sp);

#define FORTRAN_DREF_ARG17 \
   lwz r12, (ARG_OFF + 17*4)(sp); \
   lwz r12, 0(r12); \
   stw r12, (ARG_OFF + 17*4)(sp);

/*
 *  macros to get GPR arguments beyond 8
 */
#define GET_ARG8( rD )          lwz rD, (ARG_OFF + 8*4)(sp);
#define GET_ARG9( rD )          lwz rD, (ARG_OFF + 9*4)(sp);
#define GET_ARG10( rD )         lwz rD, (ARG_OFF + 10*4)(sp);
#define GET_ARG11( rD )         lwz rD, (ARG_OFF + 11*4)(sp);
#define GET_ARG12( rD )         lwz rD, (ARG_OFF + 12*4)(sp);
#define GET_ARG13( rD )         lwz rD, (ARG_OFF + 13*4)(sp);
#define GET_ARG14( rD )         lwz rD, (ARG_OFF + 14*4)(sp);
#define GET_ARG15( rD )         lwz rD, (ARG_OFF + 15*4)(sp);
#define GET_ARG16( rD )         lwz rD, (ARG_OFF + 16*4)(sp);
#define GET_ARG17( rD )         lwz rD, (ARG_OFF + 17*4)(sp);

/*
 *  macros to set GPR arguments beyond 8
 */
#define SET_ARG8( rD )          stw rD, (ARG_OFF + 8*4)(sp);
#define SET_ARG9( rD )          stw rD, (ARG_OFF + 9*4)(sp);
#define SET_ARG10( rD )         stw rD, (ARG_OFF + 10*4)(sp);
#define SET_ARG11( rD )         stw rD, (ARG_OFF + 11*4)(sp);
#define SET_ARG12( rD )         stw rD, (ARG_OFF + 12*4)(sp);
#define SET_ARG13( rD )         stw rD, (ARG_OFF + 13*4)(sp);
#define SET_ARG14( rD )         stw rD, (ARG_OFF + 14*4)(sp);
#define SET_ARG15( rD )         stw rD, (ARG_OFF + 15*4)(sp);
#define SET_ARG16( rD )         stw rD, (ARG_OFF + 16*4)(sp);
```

salppc.inc 3/9/2001

```
#define SET_ARG17( rD )                 stw rD, (ARG_OFF + 17*4)(sp);

/*
 *  macro to branch from one entry point to another
 */

#define BR FUNC( func_name ) \
   b func_name;

/*
 *  macros to call functions
 */
#define CALL FUNC( func_name ) \
   bl func_name;

#define CALL 0( func name ) \
   CALL_FUNC( func_name )

#define CALL 1( func name, arg0 ) \
   CALL_FUNC( func_name )

#define CALL 2( func name, arg0, arg1 ) \
   CALL_FUNC( func_name )

#define CALL 3( func name, arg0, arg1, arg2 ) \
   CALL_FUNC( func_name )

#define CALL 4( func name, arg0, arg1, arg2, arg3 ) \
   CALL_FUNC( func_name )

#define CALL 5( func name, arg0, arg1, arg2, arg3, arg4 ) \
   CALL_FUNC( func_name )

#define CALL 6( func name, arg0, arg1, arg2, arg3, arg4, arg5 ) \
   CALL_FUNC( func_name )

#define CALL 7( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6 ) \
   CALL_FUNC( func_name )

#define CALL 8( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7 ) \
   CALL_FUNC( func_name )

#define CALL_9( func_name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                arg8 ) \
   CALL_FUNC( func_name )

#define CALL_10( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9 ) \
   CALL_FUNC( func_name )

#define CALL_11( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10 ) \
   CALL_FUNC( func_name )

#define CALL_12( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11 ) \
   CALL_FUNC( func_name )

#define CALL_13( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12 ) \
   CALL_FUNC( func_name )

#define CALL_14( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                 arg8, arg9, arg10, arg11, arg12, arg13 ) \
   CALL_FUNC( func_name )

#define CALL_15( func_name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
```

salppc.inc 3/9/2001

```
                      arg8, arg9, arg10, arg11, arg12, arg13, arg14 ) \
    CALL_FUNC( func_name )

#define CALL_16( func name, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, \
                      arg8, arg9, arg10, arg11, arg12, arg13, arg14, arg15 ) \
    CALL_FUNC( func_name )

#if defined( BUILD MAX )
#if defined( COMPILE_ESAL_JUMP_TABLE )

/*
 *  G4 macros to create an ESAL jump table for 1, 2, 3 and 4 vector
 *  algorithms. The table name is <root_name>_jump and is made a
 *  local symbol. (not supported in C)
 */
#define DECLARE VMX_V1( root_name ) \
.section .rodata; \
.align 5; \
CONCAT( root name,  jump ): \
.long  CONCAT( root name,  n ); \
.long  CONCAT( root_name, _c );

#define DECLARE VMX_V2( root_name ) \
.section .rodata; \
.align 5; \
CONCAT( root name,  jump ): \
.long  CONCAT( root name,  nn ); \
.long  CONCAT( root name,  nc ); \
.long  CONCAT( root name,  cn ); \
.long  CONCAT( root_name, _cc );

#define DECLARE VMX_V3( root_name ) \
.section .rodata; \
.align 5; \
CONCAT( root name,  jump ): \
.long  CONCAT( root name,  nnn ); \
.long  CONCAT( root name,  nnc ); \
.long  CONCAT( root name,  ncn ); \
.long  CONCAT( root name,  ncc ); \
.long  CONCAT( root name,  cnn ); \
.long  CONCAT( root name,  cnc ); \
.long  CONCAT( root name,  ccn ); \
.long  CONCAT( root_name, _ccc );

#define DECLARE VMX_V4( root_name ) \
.section .rodata; \
.align 5; \
CONCAT( root name,  jump ): \
.long  CONCAT( root name,  nnnn ); \
.long  CONCAT( root name,  nnnc ); \
.long  CONCAT( root name,  nncn ); \
.long  CONCAT( root name,  nncc ); \
.long  CONCAT( root name,  ncnn ); \
.long  CONCAT( root name,  ncnc ); \
.long  CONCAT( root name,  nccn ); \
.long  CONCAT( root name,  nccc ); \
.long  CONCAT( root name,  cnnn ); \
.long  CONCAT( root name,  cnnc ); \
.long  CONCAT( root name,  cncn ); \
.long  CONCAT( root name,  cncc ); \
.long  CONCAT( root name,  ccnn ); \
.long  CONCAT( root name,  ccnc ); \
.long  CONCAT( root name,  cccn ); \
.long  CONCAT( root_name, _cccc );

#define DECLARE VMX_V5( root_name ) \
.section .rodata; \
```

```
.align 5; \
CONCAT( root name,  jump ): \
.long  CONCAT( root name,  nnnnn ); \
.long  CONCAT( root name,  nnnnc ); \
.long  CONCAT( root name,  nnncn ); \
.long  CONCAT( root name,  nnncc ); \
.long  CONCAT( root name,  nncnn ); \
.long  CONCAT( root name,  nncnc ); \
.long  CONCAT( root name,  nnccn ); \
.long  CONCAT( root name,  nnccc ); \
.long  CONCAT( root name,  ncnnn ); \
.long  CONCAT( root name,  ncnnc ); \
.long  CONCAT( root name,  ncncn ); \
.long  CONCAT( root name,  ncncc ); \
.long  CONCAT( root name,  nccnn ); \
.long  CONCAT( root name,  nccnc ); \
.long  CONCAT( root name,  ncccn ); \
.long  CONCAT( root name,  ncccc ); \
.long  CONCAT( root name,  cnnnn ); \
.long  CONCAT( root name,  cnnnc ); \
.long  CONCAT( root name,  cnncn ); \
.long  CONCAT( root name,  cnncc ); \
.long  CONCAT( root name,  cncnn ); \
.long  CONCAT( root name,  cncnc ); \
.long  CONCAT( root name,  cnccn ); \
.long  CONCAT( root name,  cnccc ); \
.long  CONCAT( root name,  ccnnn ); \
.long  CONCAT( root name,  ccnnc ); \
.long  CONCAT( root name,  ccncn ); \
.long  CONCAT( root name,  ccncc ); \
.long  CONCAT( root name,  cccnn ); \
.long  CONCAT( root name,  cccnc ); \
.long  CONCAT( root name,  ccccn ); \
.long  CONCAT( root_name, _ccccc );

#define DECLARE VMX Z1( root name )  DECLARE VMX V1( root name )
#define DECLARE VMX Z2( root name )  DECLARE VMX V2( root name )
#define DECLARE VMX Z3( root name )  DECLARE VMX V3( root name )
#define DECLARE VMX Z4( root name )  DECLARE VMX V4( root name )
#define DECLARE_VMX_Z5( root_name )  DECLARE_VMX_V5( root_name )

/*
 *  G4 macros to branch through the <root name> jump table based on
 *  the value of the ESAL flag. (not supported in C)
 *  (uses r0 as scratch and destroys eflag)
 *  (not supported in C)
 */
#define BR ESAL_JUMP TABLE_COMMON( root name, rtemp ) \
   addis rtemp, 0, CONCAT( root name,  jump@ha ); \
   addi rtemp, rtemp, CONCAT( root_name, _jump@l ); \
   lwzx rtemp, rtemp, r0; \
   mtctr rtemp; \
   bctr;

#define BR VMX V1( root_name, eflag, rtemp ) \
   rlwinm r0, eflag, 2, 29, 29; \
   BR_ESAL_JUMP_TABLE_COMMON( root_name, rtemp )

#define BR VMX V2( root_name, eflag, rtemp ) \
   rlwinm r0, eflag, 2, 28, 29; \
   BR_ESAL_JUMP_TABLE_COMMON( root_name, rtemp )

#define BR VMX V3( root_name, eflag, rtemp ) \
   rlwinm r0, eflag, 2, 27, 29; \
   BR_ESAL_JUMP_TABLE_COMMON( root_name, rtemp )

#define BR_VMX_V4( root_name, eflag, rtemp ) \
```

```
   rlwinm r0, eflag, 2, 26, 29; \
   BR_ESAL_JUMP_TABLE_COMMON( root_name, rtemp )

#define BR_VMX_V5( root_name, eflag, rtemp ) \
   rlwinm r0, eflag, 2, 25, 29; \
   BR_ESAL_JUMP_TABLE_COMMON( root_name, rtemp )

#define BR_VMX_Z1( root_name, eflag, rtemp ) \
        BR_VMX_V1( root_name, eflag, rtemp )

#define BR_VMX_Z2( root_name, eflag, rtemp ) \
        BR_VMX_V2( root_name, eflag, rtemp )

#define BR_VMX_Z3( root_name, eflag, rtemp ) \
        BR_VMX_V3( root_name, eflag, rtemp )

#define BR_VMX_Z4( root_name, eflag, rtemp ) \
        BR_VMX_V4( root_name, eflag, rtemp )

#define BR_VMX_Z5( root_name, eflag, rtemp ) \
        BR_VMX_V5( root_name, eflag, rtemp )

#else                                   /* no ESAL jump table */

/*
 *  G4 macros to create a dummy jump table.
 *  (not supported in C)
 */
#define DECLARE_VMX_V1( root_name )
#define DECLARE_VMX_V2( root_name )
#define DECLARE_VMX_V3( root_name )
#define DECLARE_VMX_V4( root_name )
#define DECLARE_VMX_V5( root_name )

#define DECLARE_VMX_Z1( root_name )
#define DECLARE_VMX_Z2( root_name )
#define DECLARE_VMX_Z3( root_name )
#define DECLARE_VMX_Z4( root_name )
#define DECLARE_VMX_Z5( root_name )

/*
 *  G4 macros to simply branch to root_name (no jump table)
 *  (not supported in C)
 */
#define BR_VMX_V1( root_name, eflag, rtemp ) \
   b root_name;

#define BR_VMX_V2( root_name, eflag , rtemp ) \
   b root_name;

#define BR_VMX_V3( root_name, eflag , rtemp ) \
   b root_name;

#define BR_VMX_V4( root_name, eflag , rtemp ) \
   b root_name;

#define BR_VMX_V5( root_name, eflag , rtemp ) \
   b root_name;

#define BR_VMX_Z1( root_name, eflag, rtemp ) \
        BR_VMX_V1( root_name, eflag, rtemp )

#define BR_VMX_Z2( root_name, eflag, rtemp ) \
        BR_VMX_V2( root_name, eflag, rtemp )

#define BR_VMX_Z3( root_name, eflag, rtemp ) \
        BR_VMX_V3( root_name, eflag, rtemp )
```

```
#define BR_VMX_Z4( root_name, eflag, rtemp ) \
        BR_VMX_V4( root_name, eflag, rtemp )

#define BR_VMX_Z5( root_name, eflag, rtemp ) \
        BR_VMX_V5( root_name, eflag, rtemp )

#endif                              /* end COMPILE_ESAL_JUMP_TABLE */

/*
 *  G4 macros to decide whether to enter a VMX loop
 *  VMX loop is entered if at least minimum count,
 *  all vectors have the same relative alignment
 *  (i.e., same lower 4 bits) and all strides are unit.
 *  Note, a unit_s_imm argument is provided because some
 *  packed interleaved complex functions (stride 2) such
 *  as cvaddx() can be implemented with a VMX loop.
 *  Only one macro should be invoked per source file.
 *  (uses r0 as scratch)
 *  (not supported in C)
 */
#define BR_IF_VMX_V1( root_name, min_n_imm, unit_s_imm, p1, s1, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   BR_VMX_V1( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V1_ALIGNED( root_name, min_n_imm, unit_s_imm, \
                              p1, s1, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   andi. r0, p1, 0xf; \
   bne v_skip_vmx; \
   BR_VMX_V1( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V2( root_name, min_n_imm, unit_s_imm, \
                      p1, s1, p2, s2, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s2, unit_s_imm; \
   xor r0, p1, p2; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   bne v_skip_vmx; \
   BR_VMX_V2( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V2_LS( root_name, min_n_imm, unit_s_imm, \
                         p1, s1, ps, s2, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   srwi r0, p1, 1; \
   bne v_skip_vmx; \
   cmpwi s2, unit_s_imm; \
   xor r0, r0, ps; \
   bne v_skip_vmx; \
   andi. r0, r0, 0x6; \
   bne v_skip_vmx; \
   BR_VMX_V2( root_name, eflag, s1 ) \
```

salppc.inc 3/9/2001

```
v_skip_vmx:

#define BR_IF_VMX_V2_LC( root_name, min_n_imm, unit_s_imm, \
                        p1, s1, pc, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   andi. r0, pc, 1; \
   bne v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   srwi r0, p1, 2; \
   bne v_skip_vmx; \
   xor r0, r0, pc; \
   andi. r0, r0, 0x3; \
   bne v_skip_vmx; \
   BR_VMX_V2( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V2_ALIGNED( root_name, min_n_imm, unit_s_imm, \
                             p1, s1, p2, s2, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s2, unit_s_imm; \
   or r0, p1, p2; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   bne v_skip_vmx; \
   BR_VMX_V2( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V3( root_name, min_n_imm, unit_s_imm, \
                     p1, s1, p2, s2, p3, s3, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s2, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s3, unit_s_imm; \
   xor r0, p1, p2; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   xor r0, p1, p3; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   bne v_skip_vmx; \
   BR_VMX_V3( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V3_ALIGNED( root_name, min_n_imm, unit_s_imm, \
                             p1, s1, p2, s2, p3, s3, n, eflag ) \
   cmplwi n, min_n_imm; \
   blt v_skip_vmx; \
   cmpwi s1, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s2, unit_s_imm; \
   bne v_skip_vmx; \
   cmpwi s3, unit_s_imm; \
   or r0, p1, p2; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   or r0, p1, p3; \
   bne v_skip_vmx; \
   andi. r0, r0, 0xf; \
   bne v_skip_vmx; \
   BR_VMX_V3( root_name, eflag, s1 ) \
```

```
v_skip_vmx:

#define BR_IF_VMX_V4( root name, min n imm, unit s imm, \
                      p1, s1, p2, s2, p3, s3, p4, s4, n, eflag ) \
   cmplwi n, min n_imm; \
   blt v_skip vmx; \
   cmpwi s1, unit s imm; \
   bne v_skip vmx; \
   cmpwi s2, unit s imm; \
   bne v_skip vmx; \
   cmpwi s3, unit s imm; \
   bne v_skip vmx; \
   cmpwi s4, unit s imm; \
   xor r0, p1, p2; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, p1, p3; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, p1, p4; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   bne v skip_vmx; \
   BR VMX V4( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V4_ALIGNED( root name, min n imm, unit s imm, \
                              p1, s1, p2, s2, p3, s3, p4, s4, n, eflag ) \
   cmplwi n, min n_imm; \
   blt v_skip vmx; \
   cmpwi s1, unit s imm; \
   bne v_skip vmx; \
   cmpwi s2, unit s imm; \
   bne v_skip vmx; \
   cmpwi s3, unit s imm; \
   bne v_skip vmx; \
   cmpwi s4, unit_s_imm; \
   or r0, p1, p2; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   or r0, p1, p3; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   or r0, p1, p4; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   bne v skip_vmx; \
   BR VMX V4( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V5( root name, min n imm, unit s imm, \
                      p1, s1, p2, s2, p3, s3, p4, s4, p5, s5, n, eflag ) \
   cmplwi n, min n_imm; \
   blt v_skip vmx; \
   cmpwi s1, unit s imm; \
   bne v_skip vmx; \
   cmpwi s2, unit s imm; \
   bne v_skip vmx; \
   cmpwi s3, unit s imm; \
   bne v_skip vmx; \
   cmpwi s4, unit s imm; \
   bne v_skip vmx; \
   cmpwi s5, unit s imm; \
   xor r0, p1, p2; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, p1, p3; \
   bne v_skip vmx; \
```

```
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, p1, p4; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, p1, p5; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    bne v skip_vmx; \
    BR VMX V5( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_V5_ALIGNED( root_name, min n_imm, unit s_imm, \
                              p1, s1, p2, s2, p3, s3, p4, s4, p5, s5, n, eflag )
                              \
    cmplwi n, min n_imm; \
    blt v_skip vmx; \
    cmpwi s1, unit s imm; \
    bne v_skip vmx; \
    cmpwi s2, unit s imm; \
    bne v_skip vmx; \
    cmpwi s3, unit s imm; \
    bne v_skip vmx; \
    cmpwi s4, unit s imm; \
    bne v_skip vmx; \
    cmpwi s5, unit_s_imm; \
    or r0, p1, p2; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    or r0, p1, p3; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    or r0, p1, p4; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    or r0, p1, p5; \
    bne v_skip vmx; \
    andi. r0, r0, 0xf; \
    bne v skip_vmx; \
    BR VMX V5( root_name, eflag, s1 ) \
v_skip_vmx:

#define BR_IF_VMX_Z1( root_name, min n_imm, unit_s_imm, \
                      pr1, pi1, s1, n, eflag ) \
    cmplwi n, min n_imm; \
    blt z_skip vmx; \
    cmpwi s1, unit s imm; \
    xor r0, pr1, pi2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    bne z skip_vmx; \
    BR VMX Z1( root_name, eflag, s1 ) \
z_skip_vmx:

#define BR_IF_VMX_Z2( root_name, min n imm, unit s imm, \
                      pr1, pi1, s1, pr2, pi2, s2, n, eflag ) \
    cmplwi n, min n_imm; \
    blt z_skip vmx; \
    cmpwi s1, unit s imm; \
    bne z_skip vmx; \
    cmpwi s2, unit s imm; \
    xor r0, pr1, pi1; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
```

```
    xor r0, pr1, pi2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    bne z skip_vmx; \
    BR VMX Z2( root_name, eflag, s1 ) \
z_skip_vmx:

#define BR_IF_VMX_Z3( root_name, min n imm, unit s imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, n, eflag ) \
    cmplwi n, min n_imm; \
    blt z_skip vmx; \
    cmpwi s1, unit s imm; \
    bne z_skip vmx; \
    cmpwi s2, unit s imm; \
    bne z_skip vmx; \
    cmpwi s3, unit s imm; \
    xor r0, pr1, pi1; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pi2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr3; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pi3; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    bne z skip_vmx; \
    BR VMX Z3( root_name, eflag, s1 ) \
z_skip_vmx:

#define BR_IF_VMX_Z4( root_name, min n imm, unit s imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, \
                      pr4, pi4, s4, n, eflag ) \
    cmplwi n, min n_imm; \
    blt z_skip vmx; \
    cmpwi s1, unit s imm; \
    bne z_skip vmx; \
    cmpwi s2, unit s imm; \
    bne z_skip vmx; \
    cmpwi s3, unit s imm; \
    bne z_skip vmx; \
    cmpwi s4, unit s imm; \
    xor r0, pr1, pi1; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pi2; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr3; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pi3; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pr4; \
    bne z_skip vmx; \
    andi. r0, r0, 0xf; \
    xor r0, pr1, pi4; \
    bne z_skip_vmx; \
```

```
   andi. r0, r0, 0xf; \
   bne z skip_vmx; \
   BR VMX Z4( root_name, eflag, s1 ) \
z_skip_vmx:

#define BR_IF_VMX_Z5( root_name, min n imm, unit s imm, \
                      pr1, pi1, s1, pr2, pi2, s2, pr3, pi3, s3, \
                      pr4, pi4, s4, pr5, pi5, s5, n, eflag ) \
   cmplwi n, min n_imm; \
   blt z_skip vmx; \
   cmpwi s1, unit s imm; \
   bne z_skip vmx; \
   cmpwi s2, unit s imm; \
   bne z_skip vmx; \
   cmpwi s3, unit s imm; \
   bne z_skip vmx; \
   cmpwi s4, unit s imm; \
   bne z_skip vmx; \
   cmpwi s5, unit s imm; \
   xor r0, pr1, pi1; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr2; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pi2; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr3; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pi3; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr4; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pi4; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr5; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pi5; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   bne z skip_vmx; \
   BR VMX Z5( root_name, eflag, s1 ) \
z_skip_vmx:

#define BR_IF_VMX_CONV( root name, min n imm, \
                        p1, s1, s2, p3, s3, n, eflag ) \
   cmplwi n, min n_imm; \
   blt v_skip vmx; \
   cmpwi s1, 1; \
   bne v_skip vmx; \
   cmpwi s2, 1; \
   beq PC OFFSET( 12 ); \
   cmpwi s2, -1; \
   bne v_skip vmx; \
   cmpwi s3, 1; \
   xor r0, p1, p3; \
   bne v_skip vmx; \
   andi. r0, r0, 0xf; \
   bne v skip_vmx; \
   BR VMX V3( root_name, eflag, s1 ) \
v_skip_vmx:
```

```
#define BR_IF_VMX_ZCONV( root_name, min n imm, \
                         pr1, pi1, s1, s2, pr3, pi3, s3, n, eflag ) \
   cmplwi n, min n_imm; \
   blt z_skip vmx; \
   cmpwi s1, 1; \
   bne z_skip vmx; \
   cmpwi s2, 1; \
   beq PC OFFSET( 12 ); \
   cmpwi s2, -1; \
   bne v_skip vmx; \
   cmpwi s3, 1; \
   xor r0, pr1, pi1; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pr3; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   xor r0, pr1, pi3; \
   bne z_skip vmx; \
   andi. r0, r0, 0xf; \
   bne z skip_vmx; \
   BR VMX Z3( root_name, eflag, s1 ) \
z_skip_vmx:

/*
 * G4 macro to get VMX unaligned word (FP) count
 * assumes that the last 2 bits of ptr are 0
 * sets condition code CR0
 */
#define GET VMX UNALIGNED_COUNT( count, ptr ) \
   neg count, ptr; \
   rlwinm. count, count, 30, 30, 31;

/*
 * G4 macro to get VMX unaligned short count
 * assumes that the last bit of ptr is 0
 * sets condition code CR0
 */
#define GET VMX UNALIGNED_COUNT_S( count, ptr ) \
   neg count, ptr; \
   rlwinm. count, count, 31, 29, 31;

/*
 * G4 macro to get VMX unaligned char count
 * sets condition code CR0
 */
#define GET VMX UNALIGNED_COUNT_C( count, ptr ) \
   neg count, ptr; \
   rlwinm. count, count, 0, 28, 31;

/*
 * G4 macro to load and splat an FP scalar independent of alignment
 */

#if defined( LITTLE ENDIAN )
#define SCALAR_SPLAT( vt, vtmp, scalarp ) \
   lvxl vt, 0, scalarp; \
   lvsr vtmp, 0, scalarp; \
   vperm vt, vt, vt, vtmp; \
   vspltw vt, vt, 3;
#else
#define SCALAR_SPLAT( vt, vtmp, scalarp ) \
   lvxl vt, 0, scalarp; \
   lvsl vtmp, 0, scalarp; \
   vperm vt, vt, vt, vtmp; \
   vspltw vt, vt, 0;
```

```
#endif

/*
 * G4 macro to construct an FP absolute value mask that can be used with
 * vand to take the absolute value of 4 FP numbers in a vector register
 * vt = 0x7fffffff7fffffff7fffffff7fffffff
 */
#define MAKE_VABS_MASK( vt ) \
  vspltisw vt, -1; \
  vslw vt, vt, vt; \
  vnor vt, vt, vt;

/*
 * G4 macro to construct an FP sign mask that can be used with:
 *    vandc to take the absolute value of
 *    vor to take the negative absolute value of
 *    vxor to negate
 * 4 FP numbers in a vector register
 * vt = 0x80000000800000008000000080000000
 */
#define MAKE_VSIGN_MASK( vt ) \
  vspltisw vt, -1; \
  vslw vt, vt, vt;

/*
 *  G4 macros to construct a coded touch stream control register
 *     "I" indicates argument is passed as an immediate value
 *     "R" indicates argument is passed in an integer register
 *
 *     bytes_per block = # of bytes in each block
 *        (0 = 512, 16, 32, ..., 480, 512)
 *     block count = # of blocks (0 = 256, 1, 2, 3, ... 256)
 *     byte stride = signed byte stride between start of adjacent blocks
 *        (-32768 <= byte_stride < 0; 0 = 32768; 0 < byte_stride < 32768)
 */
#define MAKE_STREAM_CODE_III( rB, bytes_per_block, block_count, byte_stride )
\
  lis rB, ((((bytes per block) >> 4) & 31) << 8) | ((block_count) & 255); \
  ori rB, rB, ((byte_stride) & 0x0000ffff);

#define MAKE STREAM CODE( rB, bytes per block, block count, byte stride ) \
        MAKE_STREAM_CODE_III( rB, bytes_per_block, block_count, byte_stride )

#define MAKE_STREAM_CODE_IIR( rB, bytes_per_block, block_count, byte_stride )
\
  lis rB, ((((bytes per block) >> 4) & 31) << 8) | ((block_count) & 255); \
  rlwimi rB, byte_stride, 0, 16, 31;

#define MAKE_STREAM_CODE_IRI( rB, bytes_per_block, block_count, byte_stride )
\
  rlwinm rB, block count, 16, 8, 15; \
  oris rB, rB, ((((bytes per_block) >> 4) & 31) << 8); \
  ori rB, rB, ((byte_stride) & 0x0000ffff);

#define MAKE_STREAM_CODE_IRR( rB, bytes_per_block, block_count, byte_stride )
\
  rlwinm rB, block count, 16, 8, 15; \
  oris rB, rB, ((((bytes per_block) >> 4) & 31) << 8); \
  rlwimi rB, byte_stride, 0, 16, 31;

#define MAKE_STREAM_CODE_RII( rB, bytes_per_block, block_count, byte_stride )
\
  rlwinm rB, bytes per block, 20, 3, 7; \
  oris rB, rB, ((block count) & 255); \
  ori rB, rB, ((byte_stride) & 0x0000ffff);

#define MAKE_STREAM_CODE_RIR( rB, bytes_per_block, block_count, byte_stride )
```

salppc.inc 3/9/2001

```
\
  rlwinm rB, bytes per block, 20, 3, 7; \
  oris rB, rB, ((block count) & 255); \
  rlwimi rB, byte_stride, 0, 16, 31;

#define MAKE_STREAM_CODE_RRI( rB, bytes_per_block, block_count, byte_stride )
\
  rlwinm rB, bytes per block, 20, 3, 7; \
  rlwimi rB, block count, 16, 8, 15; \
  ori rB, rB, ((byte_stride) & 0x0000ffff);

#define MAKE_STREAM_CODE_RRR( rB, bytes_per_block, block_count, byte_stride )
\
  rlwinm rB, bytes per block, 20, 3, 7; \
  rlwimi rB, block count, 16, 8, 15; \
  rlwimi rB, byte_stride, 0, 16, 31;

#endif                                  /* end BUILD_MAX */

#define CACHE TB THRESHOLD  1           /* 2 TB ticks = 12 CPU 100 MHz clks */
#define INSTRUCTION CACHE COUNT  3      /* min. to fully cache instructions */
#define POSTING_BUFFER_COUNT  10        /* min. to fill posting buffer      */

/*
 *  macros to set DCBx conditions explicitly
 */
#define DCBT TRUE( cond_bit, scratch ) \
   li scratch, 0; \
   cmplwi (cond_bit), scratch, 1;

#define DCBZ TRUE( cond_bit, scratch ) \
   DCBT_TRUE( cond_bit, scratch )

#define DCBT FALSE( cond_bit, scratch ) \
   li scratch, 2; \
   cmplwi (cond_bit), scratch, 1;

#define DCBZ FALSE( cond_bit, scratch ) \
   DCBT_FALSE( cond_bit, scratch )

/*
 * This macro will cause a file not to assemble.
 */
#define DO_NOT_ASSEMBLE   add scratch1, scratch2, 256;

/*
 *  Obsolete macro will cause assembler error
 */
#define TEST IF CACHABLE( cond_bit, buffer, scratch1, scratch2 ) \
      DO_NOT_ASSEMBLE
/*
 *  Obsolete macro will cause assembler error
 */

#define TEST IF CACHABLE_ALIGN( cond_bit, buffer, scratch1, scratch2 ) \
      DO_NOT_ASSEMBLE
/*
 *  macros to test if a DCBT or DCBZ instruction should be performed on
 *  a particular buffer based on a bit test (cache bit) on a specified
 *  ESAL flag.
 */

#define TEST_IF_DCBT( cond_bit, cache_bit, eflag, bufer, scratch1, scratch2 )
\
      DO_NOT_ASSEMBLE

#define SET_DCBT_COND( cond_bit, cache_bit, eflag, scratch1 ) \
```

```
   andi. scratch1, eflag, (cache bit); \
   cmplwi (cond_bit), scratch1, 0;

/*
 * Set 2 dcbt conditions and ensure only one is true
 *
 *    Ins. 1-3  Set both conditions to "No DCBT"
 *    Ins. 4    See if vec1 has a C
 *    Ins. 5    Set DCBT cond1
 *    Ins. 6    Branch if "DCBT TRUE" (eflag & bit1 = 0)
 *    Ins. 7-8  Set DCBT cond2
 */
#define SET_2_DCBT_COND( cond1 bit, cache_bit1, cond2_bit, cache_bit2, \
                         eflag, scratch ) \
   li scratch, 2; \
   cmplwi (cond1 bit), scratch, 1; \
   cmplwi (cond2 bit), scratch, 1; \
   andi. scratch, eflag, (cache_bit1); \
   cmplwi (cond1 bit), scratch, 0; \
   bc 12, ((cond1_bit)<<2)+2, PC OFFSET( 12 ); \
   andi. scratch, eflag, (cache_bit2); \
   cmplwi (cond2_bit), scratch, 0;

/*
 * Set 3 dcbt conditions and ensure only one is true
 *
 * Logic is the similar to SET_2_DCBT_COND() macro
 */
#define SET_3_DCBT_COND( cond1 bit, cache bit1, cond2 bit, cache_bit2, \
                         cond3_bit, cache_bit3, eflag, scratch ) \
   li scratch, 2; \
   cmplwi (cond1 bit), scratch, 1; \
   cmplwi (cond2 bit), scratch, 1; \
   cmplwi (cond3 bit), scratch, 1; \
   andi. scratch, eflag, (cache_bit3); \
   cmplwi (cond3 bit), scratch, 0; \
   bc 12, ((cond3_bit)<<2)+2, PC OFFSET( 24 ); \
   andi. scratch, eflag, (cache_bit2); \
   cmplwi (cond2 bit), scratch, 0; \
   bc 12, ((cond2_bit)<<2)+2, PC OFFSET( 12 ); \
   andi. scratch, eflag, (cache_bit1); \
   cmplwi (cond1_bit), scratch, 0;

/*
 * Set 4 dcbt conditions and ensure only one is true
 *
 * Logic is the similar to SET_2_DCBT_COND() macro
 */
#define SET_4_DCBT_COND( cond1 bit, cache bit1, cond2 bit, cache bit2, \
                         cond3 bit, cache_bit3, cond4_bit, cache_bit4, \
                         eflag, scratch ) \
   li scratch, 2; \
   cmplwi (cond1 bit), scratch, 1; \
   cmplwi (cond2 bit), scratch, 1; \
   cmplwi (cond3 bit), scratch, 1; \
   cmplwi (cond4 bit), scratch, 1; \
   andi. scratch, eflag, (cache_bit4); \
   cmplwi (cond4 bit), scratch, 0; \
   bc 12, ((cond4_bit)<<2)+2, PC OFFSET( 36 ); \
   andi. scratch, eflag, (cache_bit3); \
   cmplwi (cond3 bit), scratch, 0; \
   bc 12, ((cond3_bit)<<2)+2, PC OFFSET( 24 ); \
   andi. scratch, eflag, (cache_bit2); \
   cmplwi (cond2 bit), scratch, 0; \
   bc 12, ((cond2_bit)<<2)+2, PC OFFSET( 12 ); \
   andi. scratch, eflag, (cache_bit1); \
   cmplwi (cond1_bit), scratch, 0;
```

```
#if !defined COMPILE_NO_DCBZ

#define SET_DCBZ_COND( cond bit, cache bit, eflag, buffer, stride, \
                       unit stride, count, tmp1, tmp2, tmp3) \
   andi. tmp3, eflag, (cache bit); \
   cmplwi (cond bit), tmp3, 0; \
   bne PC_OFFSET( 104 ); \
   cmplwi 1, stride, unit stride; \
   bne 1, PC_OFFSET( 92 ); \
   cmplwi 1, count, (CACHE_LINE_LSIZE<<unit_stride); \
   blt 1, PC OFFSET( 84 ); \
   addi tmp2, buffer, CACHE LINE SIZE; \
   li tmp3, CACHE LINE ADDR_MASK; \
   and tmp2, tmp2, tmp3; \
   mfcr tmp3; \
   stw tmp3, CR_SAVE_OFF(sp); \
   mflr tmp3; \
   stw tmp3, LR SAVE OFF(sp); \
   CREATE STACK_FRAME( 0 ) \
   mr tmp1, r3; \
   mr r3, tmp2; \
   bl ppc buf is dcbz safe; \
   DESTROY STACK FRAME \
   lwz tmp3, LR_SAVE_OFF(sp); \
   mtlr tmp3; \
   lwz tmp3, CR_SAVE_OFF(sp); \
   mtcr tmp3; \
   li tmp2, 0; \
   cmplw 1, tmp2, r3; \
   mr r3, tmp1; \
   bne 1, PC OFFSET( 8 ); \
   cmpwi (cond_bit), count, -1;

#define SET_DCBZ_ALIGN_COND( cond bit, cache bit, eflag, buffer, stride, \
                             unit stride, count, tmp1, tmp2, tmp3) \
   andi. tmp3, eflag, (cache bit); \
   cmplwi (cond bit), tmp3, 0; \
   bne PC_OFFSET( 100 ); \
   cmplwi 1, stride, unit stride; \
   bne 1, PC_OFFSET( 88 ); \
   cmplwi 1, count, (CACHE_LINE_LSIZE<<unit_stride); \
   blt 1, PC OFFSET( 80 ); \
   andi. tmp3, buffer, CACHE_LINE_MASK; \
   bne PC OFFSET( 72 ); \
   mfcr tmp3; \
   stw tmp3, CR_SAVE_OFF(sp); \
   mflr tmp3; \
   stw tmp3, LR SAVE OFF(sp); \
   CREATE STACK_FRAME( 0 ) \
   mr tmp1, r3; \
   mr r3, buffer; \
   bl ppc buf is dcbz safe; \
   DESTROY STACK FRAME \
   lwz tmp3, LR_SAVE_OFF(sp); \
   mtlr tmp3; \
   lwz tmp3, CR_SAVE_OFF(sp); \
   mtcr tmp3; \
   li tmp2, 0; \
   cmplw 1, tmp2, r3; \
   mr r3, tmp1; \
   bne 1, PC OFFSET( 8 ); \
   cmpwi (cond_bit), count, -1;

#else /* COMPILE_NO_DCBZ is defined */

#define SET_DCBZ_COND( cond_bit, cache_bit, eflag, buffer, stride, \
```

```
                        unit stride, count, tmp1, tmp2, tmp3) \
   DCBZ_FALSE( cond_bit, tmp1 )

#define SET_DCBZ_ALIGN_COND( cond bit, cache bit, eflag, buffer, stride, \
                              unit_stride, count, tmp1, tmp2, tmp3) \
   DCBZ_FALSE( cond_bit, tmp1 )

#endif /* COMPILE_NO_DCBZ */

/*
 *  macro to perform [or skip] a dcbt instruction based on the result
 *  of a prior call to TEST IF DCBT (specifying the same condition bit).
 *  dcbt is performed if the cond "<=" is true; otherwise dcbt is skipped.
 */
#define DCBT IF( cond bit, rA, rB ) \
   bc 12, ((cond_bit)<<2)+1, PC_OFFSET( 8 ); \
   dcbt rA, rB;

/*
 *  macro to perform [or skip] a dcbz instruction based on the result
 *  of a prior call to TEST IF DCBZ (specifying the same condition bit).
 *  dcbz is performed if the cond "<=" is true; otherwise dcbz is skipped.
 */
#if !defined COMPILE_NO_DCBZ

#define DCBZ IF( cond bit, rA, rB ) \
   bc 12, ((cond_bit)<<2)+1, PC_OFFSET( 8 ); \
   dcbz rA, rB;

#else

#define DCBZ IF( cond bit, rA, rB ) \
   bc 12, ((cond_bit)<<2)+1, PC_OFFSET( 8 ); \
   nop;

#endif

/*
 *  macro to branch to a label if the buffer specified in a prior
 *  call to TEST_IF CACHABLE (also specifying the same condition bit)
 *  was cachable (i.e. TB read time was <= CACHE_TB_THRESHOLD).
 */
#define BR IF COND TRUE( cond bit, label ) \
   bc 4, ((cond_bit)<<2)+1, label;              /* <= */

/*
 *  macro to branch to a label if the buffer specified in a prior
 *  call to TEST IF CACHABLE (also specifying the same condition bit)
 *  was NOT cachable (i.e. TB read time was > CACHE_TB_THRESHOLD).
 */
#define BR IF COND FALSE( cond bit, label ) \
   bc 12, ((cond_bit)<<2)+1, label;             /* > */

/*
 *  ASIC macros
 */
#if defined( COMPILE_PREFETCH )

#define LOAD PREFETCH_CONTROL( mode, scratch1, scratch2 ) \
   li scratch1, mode; \
   addis scratch2, 0, PREFETCH CONTROL H; \
   stw scratch1, PREFETCH_CONTROL_L( scratch2 );

#define LOAD MISCON B( mode, scratch1, scratch2 ) \
   li scratch1, mode; \
   addis scratch2, 0, MISCON_B H; \
   stw scratch1, MISCON_B_L( scratch2 );
```

```
#define RESET PREFETCH CONTROL( scratch1, scratch2 ) \
   addis scratch2, 0, ASIC H; \
   lwz scratch1, MISCON B L( scratch2 ); \
   andi. scratch1, scratch1, PREFETCH MASK; \
   ori scratch1, scratch1, USE PREFETCH CONTROL; \
   stw scratch1, PREFETCH_CONTROL_L( scratch2 );

#else

#define LOAD PREFETCH CONTROL( mode, scratch1, scratch2 )
#define LOAD MISCON B( mode, scratch1, scratch2 )
#define RESET_PREFETCH_CONTROL( scratch1, scratch2 )

#endif

/*
 *  instruction macros
 */
#define ADD( rD, rA, rB )                      add rD, rA, rB;
#define ADD C( rD, rA, rB )                    add. rD, rA, rB;
#define ADDI( rD, rA, SIMM )                   addi rD, rA, (SIMM);
#define ADDIC C( rD, rA, SIMM )                addic. rD, rA, (SIMM);
#define ADDIS( rD, rA, SIMM )                  addis rD, rA, (SIMM);
#define AND( rA, rS, rB )                      and rA, rS, rB;
#define AND C( rA, rS, rB )                    and. rA, rS, rB;
#define ANDC( rA, rS, rB )                     andc rA, rS, rB;
#define ANDC C( rA, rS, rB )                   andc. rA, rS, rB;
#define ANDI C( rA, rS, UIMM )                 andi. rA, rS, (UIMM);
#define ANDIS C( rA, rS, UIMM )                andis. rA, rS, (UIMM);
#define BA( label )                            ba label;
#define BCTR                                   bctr;
#define BCTRL                                  bctrl;
#define BEQ( label )                           beq label;
#define BEQ PLUS( label )                      beq+ label;
#define BEQ MINUS( label )                     beq- label;
#define BEQ CR( bit, label )                   beq (bit), label;
#define BEQ CR PLUS( bit, label )              beq+ (bit), label;
#define BEQ CR_MINUS( bit, label )             beq- (bit), label;
#define BEQLR                                  beqlr;
#define BEQLR PLUS                             beqlr+;
#define BEQLR MINUS                            beqlr-;
#define BEQLR CR( bit )                        beqlr (bit);
#define BEQLR CR PLUS( bit )                   beqlr+ (bit);
#define BEQLR CR MINUS( bit )                  beqlr- (bit);
#define BGE( label )                           bge label;
#define BGE PLUS( label )                      bge+ label;
#define BGE MINUS( label )                     bge- label;
#define BGE CR( bit, label )                   bge (bit), label;
#define BGE CR PLUS( bit, label )              bge+ (bit), label;
#define BGE CR_MINUS( bit, label )             bge- (bit), label;
#define BGELR                                  bgelr;
#define BGELR PLUS                             bgelr+;
#define BGELR MINUS                            bgelr-;
#define BGELR CR( bit )                        bgelr (bit);
#define BGELR CR PLUS( bit )                   bgelr+ (bit);
#define BGELR CR MINUS( bit )                  bgelr- (bit);
#define BGT( label )                           bgt label;
#define BGT PLUS( label )                      bgt+ label;
#define BGT MINUS( label )                     bgt- label;
#define BGT CR( bit, label )                   bgt (bit), label;
#define BGT CR PLUS( bit, label )              bgt+ (bit), label;
#define BGT CR_MINUS( bit, label )             bgt- (bit), label;
#define BGTLR                                  bgtlr;
#define BGTLR PLUS                             bgtlr+;
#define BGTLR MINUS                            bgtlr-;
#define BGTLR_CR( bit )                        bgtlr (bit);
```

salppc.inc 3/9/2001

```
#define BGTLR CR PLUS( bit )                  bgtlr+ (bit);
#define BGTLR CR MINUS( bit )                 bgtlr- (bit);
#define BL( label )                           bl label;
#define BLE( label )                          ble label;
#define BLE PLUS( label )                     ble+ label;
#define BLE MINUS( label )                    ble- label;
#define BLE CR( bit, label )                  ble (bit), label;
#define BLE CR PLUS( bit, label )             ble+ (bit), label;
#define BLE CR_MINUS( bit, label )            ble- (bit), label;
#define BLELR                                 blelr;
#define BLELR PLUS                            blelr+;
#define BLELR MINUS                           blelr-;
#define BLELR CR( bit )                       blelr (bit);
#define BLELR CR PLUS( bit )                  blelr+ (bit);
#define BLELR_CR_MINUS( bit )                 blelr- (bit);
#define BLR                                   blr;
#define BLRL                                  blrl;
#define BLT( label )                          blt label;
#define BLT PLUS( label )                     blt+ label;
#define BLT MINUS( label )                    blt- label;
#define BLT CR( bit, label )                  blt (bit), label;
#define BLT CR PLUS( bit, label )             blt+ (bit), label;
#define BLT CR_MINUS( bit, label )            blt- (bit), label;
#define BLTLR                                 bltlr;
#define BLTLR PLUS                            bltlr+;
#define BLTLR MINUS                           bltlr-;
#define BLTLR CR( bit )                       bltlr (bit);
#define BLTLR CR PLUS( bit )                  bltlr+ (bit);
#define BLTLR CR MINUS( bit )                 bltlr- (bit);
#define BNE( label )                          bne label;
#define BNE PLUS( label )                     bne+ label;
#define BNE MINUS( label )                    bne- label;
#define BNE CR( bit, label )                  bne (bit), label;
#define BNE CR PLUS( bit, label )             bne+ (bit), label;
#define BNE CR_MINUS( bit, label )            bne- (bit), label;
#define BNELR                                 bnelr;
#define BNELR PLUS                            bnelr+;
#define BNELR MINUS                           bnelr-;
#define BNELR CR( bit )                       bnelr (bit);
#define BNELR CR PLUS( bit )                  bnelr+ (bit);
#define BNELR CR MINUS( bit )                 bnelr- (bit);
#define BR( label )                           b label;
#define CLRLWI( rA, rS, nbits )               clrlwi rA, rS, (nbits);
#define CLRLWI C( rA, rS, nbits )             clrlwi. rA, rS, (nbits);
#define CLRRWI( rA, rS, nbits )               clrrwi rA, rS, (nbits);
#define CLRRWI_C( rA, rS, nbits )             clrrwi. rA, rS, (nbits);
#define CMPLW( rA, rB )                       cmplw rA, rB;
#define CMPLW CR( bit, rA, rB )               cmplw bit, rA, rB;
#define CMPLWI( rA, UIMM )                    cmplwi rA, (UIMM);
#define CMPLWI CR( bit, rA, UIMM )            cmplwi bit, rA, (UIMM);
#define CMPW( rA, rB )                        cmpw rA, rB;
#define CMPW CR( bit, rA, rB )                cmpw bit, rA, rB;
#define CMPWI( rA, SIMM )                     cmpwi rA, (SIMM);
#define CMPWI_CR( bit, rA, SIMM )             cmpwi bit, rA, (SIMM);
#define DCBF( rA, rB )                        dcbf rA, rB;
#define DCBI( rA, rB )                        dcbi rA, rB;
#define DCBST( rA, rB )                       dcbst rA, rB;
#define DCBT( rA, rB )                        dcbt rA, rB;
#define DCBTST( rA, rB )                      dcbtst rA, rB;
#if !defined COMPILE_NO_DCBZ
#define DCBZ( rA, rB )                        dcbz rA, rB;
#else
#define DCBZ( rA, rB )                        nop;
#endif
#define DECR( rD )                            addi rD, rD, -1;
#define DECR C( rD )                          addic. rD, rD, -1;
#define DIVW( rD, rA, rB )                    divw rD, rA, rB;
```

```
#define DIVW C( rD, rA, rB )                divw. rD, rA, rB;
#define DIVWU( rD, rA, rB )                 divwu rD, rA, rB;
#define DIVWU C( rD, rA, rB )               divwu. rD, rA, rB;
#define EQV( rA, rS, rB )                   eqv rA, rS, rB;
#define EQV C( rA, rS, rB )                 eqv. rA, rS, rB;
#define EXTLWI( rA, rS, n, b )              rlwinm rA, rS, (b), 0, (n)-1;
#define EXTLWI C( rA, rS, n, b )            rlwinm. rA, rS, (b), 0, (n)-1;
#define EXTRWI( rA, rS, n, b )              rlwinm rA, rS, (b)+(n), 32-(n), 31;
#define EXTRWI C( rA, rS, n, b )            rlwinm. rA, rS, (b)+(n), 32-(n), 31;
#define FABS( frD, frB )                    fabs frD, frB;
#define FADD( frD, frA, frB )               fadd frD, frA, frB;
#define FADDS( frD, frA, frB )              fadds frD, frA, frB;
#define FCMPO( bit, frA, frB )              fcmpo bit, frA, frB;
#define FCMPU( bit, frA, frB )              fcmpu bit, frA, frB;
#define FCTIW( frD, frB )                   fctiw frD, frB;
#define FCTIWZ( frD, frB )                  fctiwz frD, frB;
#define FDIV( frD, frA, frB )               fdiv frD, frA, frB;
#define FDIVS( frD, frA, frB )              fdivs frD, frA, frB;
#define FMADD( frD, frA, frC, frB )         fmadd frD, frA, frC, frB;
#define FMADDS( frD, frA, frC, frB )        fmadds frD, frA, frC, frB;
#define FMOV( frD, frB )                    FMR( frD, frB )
#define FMR( frD, frB )                     fmr frD, frB;
#define FMUL( frD, frA, frB )               fmul frD, frA, frB;
#define FMULS( frD, frA, frB )              fmuls frD, frA, frB;
#define FMSUB( frD, frA, frC, frB )         fmsub frD, frA, frC, frB;
#define FMSUBS( frD, frA, frC, frB )        fmsubs frD, frA, frC, frB;
#define FNABS( frD, frB )                   fnabs frD, frB;
#define FNEG( frD, frB )                    fneg frD, frB;
#define FNMADD( frD, frA, frC, frB )        fnmadd frD, frA, frC, frB;
#define FNMADDS( frD, frA, frC, frB )       fnmadds frD, frA, frC, frB;
#define FNMSUB( frD, frA, frC, frB )        fnmsub frD, frA, frC, frB;
#define FNMSUBS( frD, frA, frC, frB )       fnmsubs frD, frA, frC, frB;
#define FRES( frD, frB )                    fres frD, frB;
#define FRSP( frD, frB )                    frsp frD, frB;
#define FRSQRTE( frD, frB )                 frsqrte frD, frB;
#define FSEL( frD, frA, frC, frB )          fsel frD, frA, frC, frB;
#define FSUB( frD, frA, frB )               fsub frD, frA, frB;
#define FSUBS( frD, frA, frB )              fsubs frD, frA, frB;
#define GOTO( label )                       BR( label )
#define INCR( rD )                          addi rD, rD, 1;
#define INCR C( rD )                        addic. rD, rD, 1;
#define INSLWI( rA, rS, n, b )              rlwimi rA, rS, 32-(b), (b), (b)+(n)-1;
#define INSLWI_C( rA, rS, n, b )            rlwimi. rA, rS, 32-(b), (b), (b)
+(n)-1;
#define INSRWI( rA, rS, n, b )              rlwimi rA, rS, 32-((b)+(n)), (b), (b)
+(n)-1;
#define INSRWI_C( rA, rS, n, b )            rlwimi. rA, rS, 32-((b)+(n)), (b), (b)
+(n)-1;
#define LA( rD, symbol, SIMM )              addis rD, 0, (symbol+(SIMM))@ha; \
                                            addi rD, rD, (symbol+(SIMM))@l;
#define LABEL( label )                      label:
#define LBZ( rD, rA, d )                    lbz rD, (d)(rA);
#define LBZA( rD, symbol )                  addis rD, 0, (symbol)@ha; \
                                            lbz rD, (symbol)@l(rD);
#define LBZU( rD, rA, d )                   lbzu rD, (d)(rA);
#define LBZUX( rD, rA, rB )                 lbzux rD, rA, rB;
#define LBZX( rD, rA, rB )                  lbzx rD, rA, rB;
#define LFD( frD, rA, d )                   lfd frD, (d)(rA);
#define LFDU( frD, rA, d )                  lfdu frD, (d)(rA);
#define LFDUX( frD, rA, rB )                lfdux frD, rA, rB;
#define LFDX( frD, rA, rB )                 lfdx frD, rA, rB;
#define LFS( frD, rA, d )                   lfs frD, (d)(rA);
#define LFSA( frD, symbol, rT )             addis rT, 0, (symbol)@ha; \
                                            lfs frD, (symbol)@l(rT);
#define LFSU( frD, rA, d )                  lfsu frD, (d)(rA);
#define LFSUX( frD, rA, rB )                lfsux frD, rA, rB;
#define LFSX( frD, rA, rB )                 lfsx frD, rA, rB;
```

```
#define LHA( rD, rA, d )                      lha rD, (d)(rA);
#define LHAA( rD, symbol )                    addis rD, 0, (symbol)@ha; \
                                              lha rD, (symbol)@l(rD);
#define LHAU( rD, rA, d )                     lhau rD, (d)(rA);
#define LHAUX( rD, rA, rB )                   lhaux rD, rA, rB;
#define LHAX( rD, rA, rB )                    lhax rD, rA, rB;
#define LHZ( rD, rA, d )                      lhz rD, (d)(rA);
#define LHZA( rD, symbol )                    addis rD, 0, (symbol)@ha; \
                                              lhz rD, (symbol)@l(rD);
#define LHZU( rD, rA, d )                     lhzu rD, (d)(rA);
#define LHZUX( rD, rA, rB )                   lhzux rD, rA, rB;
#define LHZX( rD, rA, rB )                    lhzx rD, rA, rB;
#define LI( rD, SIMM )                        li rD, (SIMM);
#define LIS( rD, SIMM )                       lis rD, (SIMM);
#define LOAD_COUNT( rD )                      mtctr rD;
#define LWZ( rD, rA, d )                      lwz rD, (d)(rA);
#define LWZA( rD, symbol )                    addis rD, 0, (symbol)@ha; \
                                              lwz rD, (symbol)@l(rD);
#define LWZU( rD, rA, d )                     lwzu rD, (d)(rA);
#define LWZUX( rD, rA, rB )                   lwzux rD, rA, rB;
#define LWZX( rD, rA, rB )                    lwzx rD, rA, rB;
#define MCRF( crfD, crfS )                    mcrf crfD, crfS;
#define MCRFS( crfD, crfS )                   mcrfs crfD, crfS;
#define MFCR( rD )                            mfcr rD;
#define MFCTR( rD )                           mfctr rD;
#define MFLR( rD )                            mflr rD;
#define MFSPR( rD, SPR )                      mfspr rD, SPR;
#define MR( rA, rS )                          mr rA, rS;
#define MR_C( rA, rS )                        or. rA, rS, rS;
#define MOV( rA, rS )                         MR( rA, rS )
#define MOV_C( rA, rS )                       MR_C( rA, rS )
#define MTCR( rD )                            mtcr rD;
#define MTCTR( rD )                           mtctr rD;
#define MTFSFI( crfD, IMM )                   mtfsfi (crfD), (IMM);
#define MTLR( rD )                            mtlr rD;
#define MTSPR( SPR, rS )                      mtspr SPR, rS;
#define MULLI( rD, rA, SIMM )                 mulli rD, rA, (SIMM);
#define MULLW( rD, rA, rB )                   mullw rD, rA, rB;
#define MULLW_C( rD, rA, rB )                 mullw. rD, rA, rB;
#define NAND( rA, rS, rB )                    nand rA, rS, rB;
#define NAND_C( rA, rS, rB )                  nand. rA, rS, rB;
#define NEG( rD, rA )                         neg rD, rA;
#define NEG_C( rD, rA )                       neg. rD, rA;
#define NOP                                   nop;
#define NOR( rA, rS, rB )                     nor rA, rS, rB;
#define NOR_C( rA, rS, rB )                   nor. rA, rS, rB;
#define OR( rA, rS, rB )                      or rA, rS, rB;
#define OR_C( rA, rS, rB )                    or. rA, rS, rB;
#define ORC( rA, rS, rB )                     orc rA, rS, rB;
#define ORC_C( rA, rS, rB )                   orc. rA, rS, rB;
#define ORI( rA, rS, UIMM )                   ori rA, rS, (UIMM);
#define ORIS( rA, rS, UIMM )                  oris rA, rS, (UIMM);
#define RETURN                                BLR
#define RLWIMI( rA, rS, SH, MB, ME )          rlwimi rA, rS, SH, MB, ME;
#define RLWIMI_C( rA, rS, SH, MB, ME )        rlwimi. rA, rS, SH, MB, ME;
#define RLWINM( rA, rS, SH, MB, ME )          rlwinm rA, rS, SH, MB, ME;
#define RLWINM_C( rA, rS, SH, MB, ME )        rlwinm. rA, rS, SH, MB, ME;
#define RLWNM( rA, rS, rB, MB, ME )           rlwnm rA, rS, rB, MB, ME;
#define RLWNM_C( rA, rS, rB, MB, ME )         rlwnm. rA, rS, rB, MB, ME;
#define ROTLW( rA, rS, rB )                   rlwnm rA, rS, rB, 0, 31;
#define ROTLW_C( rA, rS, rB )                 rlwnm. rA, rS, rB, 0, 31;
#define ROTLWI( rA, rS, n )                   rlwinm rA, rS, (n), 0, 31;
#define ROTLWI_C( rA, rS, n )                 rlwinm. rA, rS, (n), 0, 31;
#define ROTRWI( rA, rS, n )                   rlwinm rA, rS, 32-(n), 0, 31;
#define ROTRWI_C( rA, rS, n )                 rlwinm. rA, rS, 32-(n), 0, 31;
#define SLW( rA, rS, rB )                     slw rA, rS, rB;
#define SLW_C( rA, rS, rB )                   slw. rA, rS, rB;
```

```
#define SLWI( rA, rS, SH )              slwi rA, rS, (SH);
#define SLWI_C( rA, rS, SH )            slwi. rA, rS, (SH);
#define SRAW( rA, rS, rB )              sraw rA, rS, rB;
#define SRAW_C( rA, rS, rB )            sraw. rA, rS, rB;
#define SRAWI( rA, rS, SH )             srawi rA, rS, (SH);
#define SRAWI_C( rA, rS, SH )           srawi. rA, rS, (SH);
#define SRW( rA, rS, rB )               srw rA, rS, rB;
#define SRW_C( rA, rS, rB )             srw. rA, rS, rB;
#define SRWI( rA, rS, SH )              srwi rA, rS, (SH);
#define SRWI_C( rA, rS, SH )            srwi. rA, rS, (SH);
#define STB( rS, rA, d )                stb rS, (d)(rA);
#define STBU( rS, rA, d )               stbu rS, (d)(rA);
#define STBUX( rS, rA, rB )             stbux rS, rA, rB;
#define STBX( rS, rA, rB )              stbx rS, rA, rB;
#define STFD( frD, rA, d )              stfd frD, (d)(rA);
#define STFDU( frD, rA, d )             stfdu frD, (d)(rA);
#define STFDUX( frD, rA, rB )           stfdux frD, rA, rB;
#define STFDX( frD, rA, rB )            stfdx frD, rA, rB;
#define STFS( frD, rA, d )              stfs frD, (d)(rA);
#define STFSU( frD, rA, d )             stfsu frD, (d)(rA);
#define STFSUX( frD, rA, rB )           stfsux frD, rA, rB;
#define STFSX( frD, rA, rB )            stfsx frD, rA, rB;
#define STH( rS, rA, d )                sth rS, (d)(rA);
#define STHU( rS, rA, d )               sthu rS, (d)(rA);
#define STHUX( rS, rA, rB )             sthux rS, rA, rB;
#define STHX( rS, rA, rB )              sthx rS, rA, rB;
#define STW( rS, rA, d )                stw rS, (d)(rA);
#define STWU( rS, rA, d )               stwu rS, (d)(rA);
#define STWUX( rS, rA, rB )             stwux rS, rA, rB;
#define STWX( rS, rA, rB )              stwx rS, rA, rB;
#define SUB( rD, rA, rB )               sub rD, rA, rB;
#define SUB_C( rD, rA, rB )             sub. rD, rA, rB;
#define SUBFIC( rD, rA, SIMM )          subfic rD, rA, (SIMM);
#define SUBI( rD, rA, SIMM )            subi rD, rA, (SIMM);
#define SUBIC_C( rD, rA, SIMM )         subic. rD, rA, (SIMM);
#define SUBIS( rD, rA, SIMM )           subis rD, rA, (SIMM);
#define TEST_COUNT( label )             bdnz label;
#define XOR( rA, rS, rB )               xor rA, rS, rB;
#define XOR_C( rA, rS, rB )             xor. rA, rS, rB;
#define XORI( rA, rS, UIMM )            xori rA, rS, (UIMM);
#define XORIS( rA, rS, UIMM )           xoris rA, rS, (UIMM);

/*
 *  VMX instructions
 */
#define BR_VMX_ALL_TRUE( label )        bt 24, label;
#define BR_VMX_ALL_FALSE( label )       bt 26, label;
#define BR_VMX_NONE_TRUE( label )       bt 26, label;
#define BR_VMX_SOME_FALSE( label )      bf 24, label;
#define BR_VMX_SOME_TRUE( label )       bf 26, label;
#define DSS( STRM )                     dss STRM, 0;
#define DSSALL                          dss 0, 1;
#define DST( rA, rB, STRM )             dst rA, rB, STRM;
#define DSTST( rA, rB, STRM )           dstst rA, rB, STRM;
#define DSTT( rA, rB, STRM )            dstt rA, rB, STRM;
#define DSTSTT( rA, rB, STRM )          dststt rA, rB, STRM;
#define LVEBX( vT, rA, rB )             lvebx vT, rA, rB;
#define LVEHX( vT, rA, rB )             lvehx vT, rA, rB;
#define LVEWX( vT, rA, rB )             lvewx vT, rA, rB;

#if defined( LITTLE_ENDIAN )
#define LVSL( vT, rA, rB )              lvsr vT, rA, rB;
#define LVSR( vT, rA, rB )              lvsl vT, rA, rB;
#else
#define LVSL( vT, rA, rB )              lvsl vT, rA, rB;
#define LVSR( vT, rA, rB )              lvsr vT, rA, rB;
#endif
```

```
#define LVX( vT, rA, rB )                    lvx vT, rA, rB;
#define LVXL( vT, rA, rB )                   lvxl vT, rA, rB;
#define STVEBX( vS, rA, rB )                 stvebx vS, rA, rB;
#define STVEHX( vS, rA, rB )                 stvehx vS, rA, rB;
#define STVEWX( vS, rA, rB )                 stvewx vS, rA, rB;
#define STVX( vS, rA, rB )                   stvx vS, rA, rB;
#define STVXL( vS, rA, rB )                  stvxl vS, rA, rB;
#define VADDFP( vT, vA, vB )                 vaddfp vT, vA, vB;
#define VADDSBS( vT, vA, vB )                vaddsbs vT, vA, vB;
#define VADDSHS( vT, vA, vB )                vaddshs vT, vA, vB;
#define VADDSWS( vT, vA, vB )                vaddsws vT, vA, vB;
#define VADDUBM( vT, vA, vB )                vaddubm vT, vA, vB;
#define VADDUBS( vT, vA, vB )                vaddubs vT, vA, vB;
#define VADDUHM( vT, vA, vB )                vadduhm vT, vA, vB;
#define VADDUHS( vT, vA, vB )                vadduhs vT, vA, vB;
#define VADDUWM( vT, vA, vB )                vadduwm vT, vA, vB;
#define VADDUWS( vT, vA, vB )                vadduws vT, vA, vB;
#define VAND( vT, vA, vB )                   vand vT, vA, vB;
#define VANDC( vT, vA, vB )                  vandc vT, vA, vB;
#define VCMPEQFP( vT, vA, vB )               vcmpeqfp vT, vA, vB;
#define VCMPEQFP C( vT, vA, vB )             vcmpeqfp. vT, vA, vB;
#define VCMPEQUB( vT, vA, vB )               vcmpequb vT, vA, vB;
#define VCMPEQUB C( vT, vA, vB )             vcmpequb. vT, vA, vB;
#define VCMPEQUH( vT, vA, vB )               vcmpequh vT, vA, vB;
#define VCMPEQUH C( vT, vA, vB )             vcmpequh. vT, vA, vB;
#define VCMPEQUW( vT, vA, vB )               vcmpequw vT, vA, vB;
#define VCMPEQUW C( vT, vA, vB )             vcmpequw. vT, vA, vB;
#define VCMPGEFP( vT, vA, vB )               vcmpgefp vT, vA, vB;
#define VCMPGEFP C( vT, vA, vB )             vcmpgefp. vT, vA, vB;
#define VCMPGTFP( vT, vA, vB )               vcmpgtfp vT, vA, vB;
#define VCMPGTFP C( vT, vA, vB )             vcmpgtfp. vT, vA, vB;
#define VCMPGTSB( vT, vA, vB )               vcmpgtsb vT, vA, vB;
#define VCMPGTSB C( vT, vA, vB )             vcmpgtsb. vT, vA, vB;
#define VCMPGTSH( vT, vA, vB )               vcmpgtsh vT, vA, vB;
#define VCMPGTSH C( vT, vA, vB )             vcmpgtsh. vT, vA, vB;
#define VCMPGTSW( vT, vA, vB )               vcmpgtsw vT, vA, vB;
#define VCMPGTSW C( vT, vA, vB )             vcmpgtsw. vT, vA, vB;
#define VCMPGTUB( vT, vA, vB )               vcmpgtub vT, vA, vB;
#define VCMPGTUB C( vT, vA, vB )             vcmpgtub. vT, vA, vB;
#define VCMPGTUH( vT, vA, vB )               vcmpgtuh vT, vA, vB;
#define VCMPGTUH C( vT, vA, vB )             vcmpgtuh. vT, vA, vB;
#define VCMPGTUW( vT, vA, vB )               vcmpgtuw vT, vA, vB;
#define VCMPGTUW C( vT, vA, vB )             vcmpgtuw. vT, vA, vB;
#define VCFSX( vT, vB, UIMM )                vcfsx vT, vB, (UIMM);
#define VCFUX( vT, vB, UIMM )                vcfux vT, vB, (UIMM);
#define VCTSXS( vT, vB, UIMM )               vctsxs vT, vB, (UIMM);
#define VCTUXS( vT, vB, UIMM )               vctuxs vT, vB, (UIMM);
#define VEXPTEFP( vT, vB )                   vexptefp vT, vB;
#define VLOGEFP( vT, vB )                    vlogefp vT, vB;
#define VMADDFP( vT, vA, vC, vB )            vmaddfp vT, vA, vC, vB;
#define VMAXFP( vT, vA, vB )                 vmaxfp vT, vA, vB;
#define VMAXSB( vT, vA, vB )                 vmaxsb vT, vA, vB;
#define VMAXSH( vT, vA, vB )                 vmaxsh vT, vA, vB;
#define VMAXSW( vT, vA, vB )                 vmaxsw vT, vA, vB;
#define VMAXUB( vT, vA, vB )                 vmaxub vT, vA, vB;
#define VMAXUH( vT, vA, vB )                 vmaxuh vT, vA, vB;
#define VMAXUW( vT, vA, vB )                 vmaxuw vT, vA, vB;
#define VMHADDSHS( vD, vA, vB, vC )          vmhaddshs vD, vA, vB, vC;
#define VMHRADDSHS( vD, vA, vB, vC )         vmhraddshs vD, vA, vB, vC;
#define VMINFP( vT, vA, vB )                 vminfp vT, vA, vB;
#define VMINSB( vT, vA, vB )                 vminsb vT, vA, vB;
#define VMINSH( vT, vA, vB )                 vminsh vT, vA, vB;
#define VMINSW( vT, vA, vB )                 vminsw vT, vA, vB;
#define VMINUB( vT, vA, vB )                 vminub vT, vA, vB;
#define VMINUH( vT, vA, vB )                 vminuh vT, vA, vB;
#define VMINUW( vT, vA, vB )                 vminuw vT, vA, vB;
```

```
#define VMLADDUHM( vD, vA, vB, vC )     vmladduhm vD, vA, vB, vC;
#define VMR( vD, vS )                   vor vD, vS, vS;

#if defined( LITTLE_ENDIAN )
#define VMRGHB( vT, vA, vB )            vmrglb vT, vB, vA;
#define VMRGHH( vT, vA, vB )            vmrglh vT, vB, vA;
#define VMRGHW( vT, vA, vB )            vmrglw vT, vB, vA;
#define VMRGLB( vT, vA, vB )            vmrghb vT, vB, vA;
#define VMRGLH( vT, vA, vB )            vmrghh vT, vB, vA;
#define VMRGLW( vT, vA, vB )            vmrghw vT, vB, vA;
#else
#define VMRGHB( vT, vA, vB )            vmrghb vT, vA, vB;
#define VMRGHH( vT, vA, vB )            vmrghh vT, vA, vB;
#define VMRGHW( vT, vA, vB )            vmrghw vT, vA, vB;
#define VMRGLB( vT, vA, vB )            vmrglb vT, vA, vB;
#define VMRGLH( vT, vA, vB )            vmrglh vT, vA, vB;
#define VMRGLW( vT, vA, vB )            vmrglw vT, vA, vB;
#endif

#define VMSUMMBM( vT, vA, vB, vC )      vmsummbm vT, vA, vB, vC;
#define VMSUMSHM( vT, vA, vB, vC )      vmsumshm vT, vA, vB, vC;
#define VMSUMSHS( vT, vA, vB, vC )      vmsumshs vT, vA, vB, vC;
#define VMSUMUBM( vT, vA, vB, vC )      vmsumubm vT, vA, vB, vC;
#define VMSUMUHM( vT, vA, vB, vC )      vmsumuhm vT, vA, vB, vC;
#define VMSUMUHS( vT, vA, vB, vC )      vmsumuhs vT, vA, vB, vC;
#define VMULESB( vT, vA, vB )           vmulesb vT, vA, vB;
#define VMULESH( vT, vA, vB )           vmulesh vT, vA, vB;
#define VMULEUB( vT, vA, vB )           vmuleub vT, vA, vB;
#define VMULEUH( vT, vA, vB )           vmuleuh vT, vA, vB;
#define VMULOSB( vT, vA, vB )           vmulosb vT, vA, vB;
#define VMULOSH( vT, vA, vB )           vmulosh vT, vA, vB;
#define VMULOUB( vT, vA, vB )           vmuloub vT, vA, vB;
#define VMULOUH( vT, vA, vB )           vmulouh vT, vA, vB;
#define VNMSUBFP( vT, vA, vC, vB )      vnmsubfp vT, vA, vC, vB;
#define VNOR( vT, vA, vB )              vnor vT, vA, vB;
#define VNOT( vT, vA )                  vnor vT, vA, vA;
#define VOR( vT, vA, vB )               vor vT, vA, vB;

#if defined( LITTLE ENDIAN )
#define VPERM( vT, vA, vB, vC )         vperm vT, vB, vA, vC;
#define VPKUHUM( vT, vA, vB )           vpkuhum vT, vB, vA;
#define VPKUHUS( vT, vA, vB )           vpkuhus vT, vB, vA;
#define VPKSHUS( vT, vA, vB )           vpkshus vT, vB, vA;
#define VPKSHSS( vT, vA, vB )           vpkshss vT, vB, vA;
#define VPKUWUM( vT, vA, vB )           vpkuwum vT, vB, vA;
#define VPKUWUS( vT, vA, vB )           vpkuwus vT, vB, vA;
#define VPKSWUS( vT, vA, vB )           vpkswus vT, vB, vA;
#define VPKSWSS( vT, vA, vB )           vpkswss vT, vB, vA;
#else
#define VPERM( vT, vA, vB, vC )         vperm vT, vA, vB, vC;
#define VPKUHUM( vT, vA, vB )           vpkuhum vT, vA, vB;
#define VPKUHUS( vT, vA, vB )           vpkuhus vT, vA, vB;
#define VPKSHUS( vT, vA, vB )           vpkshus vT, vA, vB;
#define VPKSHSS( vT, vA, vB )           vpkshss vT, vA, vB;
#define VPKUWUM( vT, vA, vB )           vpkuwum vT, vA, vB;
#define VPKUWUS( vT, vA, vB )           vpkuwus vT, vA, vB;
#define VPKSWUS( vT, vA, vB )           vpkswus vT, vA, vB;
#define VPKSWSS( vT, vA, vB )           vpkswss vT, vA, vB;
#endif

#define VREFP( vT, vB )                 vrefp vT, vB;
#define VRFIM( vT, vB )                 vrfim vT, vB;
#define VRFIN( vT, vB )                 vrfin vT, vB;
#define VRFIP( vT, vB )                 vrfip vT, vB;
#define VRFIZ( vT, vB )                 vrfiz vT, vB;
#define VRLB( vT, vA, vB )              vrlb vT, vA, vB;
#define VRLH( vT, vA, vB )              vrlh vT, vA, vB;
```

```
#define VRLW( vT, vA, vB )              vrlw vT, vA, vB;
#define VRSQRTEFP( vT, vB )             vrsqrtefp vT, vB;
#define VSEL( vT, vA, vB, vC )          vsel vT, vA, vB, vC;
#define VSL( vT, vA, vB )               vsl vT, vA, vB;

#if defined( LITTLE_ENDIAN )
#define VSLDOI( vT, vA, vB, UIMM )      vsldoi vT, vB, vA, (16 - (UIMM));
#else
#define VSLDOI( vT, vA, vB, UIMM )      vsldoi vT, vA, vB, (UIMM);
#endif

#define VSLB( vT, vA, vB )              vslb vT, vA, vB;
#define VSLH( vT, vA, vB )              vslh vT, vA, vB;
#define VSLO( vT, vA, vB )              vslo vT, vA, vB;
#define VSLW( vT, vA, vB )              vslw vT, vA, vB;
#define VSR( vT, vA, vB )               vsr vT, vA, vB;
#define VSRAB( vT, vA, vB )             vsrab vT, vA, vB;
#define VSRAH( vT, vA, vB )             vsrah vT, vA, vB;
#define VSRAW( vT, vA, vB )             vsraw vT, vA, vB;
#define VSRB( vT, vA, vB )              vsrb vT, vA, vB;
#define VSRH( vT, vA, vB )              vsrh vT, vA, vB;
#define VSRO( vT, vA, vB )              vsro vT, vA, vB;
#define VSRW( vT, vA, vB )              vsrw vT, vA, vB;
#define VSPLTB( vT, vB, UIMM )          vspltb vT, vB, C_INDEX_MUNGE( UIMM );
#define VSPLTH( vT, vB, UIMM )          vsplth vT, vB, S_INDEX_MUNGE( UIMM );
#define VSPLTW( vT, vB, UIMM )          vspltw vT, vB, L_INDEX_MUNGE( UIMM );
#define VSPLTISB( vT, SIMM )            vspltisb vT, (SIMM);
#define VSPLTISH( vT, SIMM )            vspltish vT, (SIMM);
#define VSPLTISW( vT, SIMM )            vspltisw vT, (SIMM);
#define VSUBFP( vT, vA, vB )            vsubfp vT, vA, vB;
#define VSUBSBS( vT, vA, vB )           vsubsbs vT, vA, vB;
#define VSUBSHS( vT, vA, vB )           vsubshs vT, vA, vB;
#define VSUBSWS( vT, vA, vB )           vsubsws vT, vA, vB;
#define VSUBUBM( vT, vA, vB )           vsububm vT, vA, vB;
#define VSUBUBS( vT, vA, vB )           vsububs vT, vA, vB;
#define VSUBUHM( vT, vA, vB )           vsubuhm vT, vA, vB;
#define VSUBUHS( vT, vA, vB )           vsubuhs vT, vA, vB;
#define VSUBUWM( vT, vA, vB )           vsubuwm vT, vA, vB;
#define VSUBUWS( vT, vA, vB )           vsubuws vT, vA, vB;
#define VSUMSWS( vT, vA, vB )           vsumsws vT, vA, vB;
#define VSUM2SWS( vT, vA, vB )          vsum2sws vT, vA, vB;
#define VSUM4SBS( vT, vA, vB )          vsum4sbs vT, vA, vB;
#define VSUM4SHS( vT, vA, vB )          vsum4shs vT, vA, vB;
#define VSUM4UBS( vT, vA, vB )          vsum4ubs vT, vA, vB;

#if defined( LITTLE_ENDIAN )
#define VUPKHSB( vT, vB )               vupklsb vT, vB;
#define VUPKHSH( vT, vB )               vupklsh vT, vB;
#define VUPKLSB( vT, vB )               vupkhsb vT, vB;
#define VUPKLSH( vT, vB )               vupkhsh vT, vB;
#else
#define VUPKHSB( vT, vB )               vupkhsb vT, vB;
#define VUPKHSH( vT, vB )               vupkhsh vT, vB;
#define VUPKLSB( vT, vB )               vupklsb vT, vB;
#define VUPKLSH( vT, vB )               vupklsh vT, vB;
#endif

#define VXOR( vT, vA, vB )              vxor vT, vA, vB;

/*
 *  stack and register macros
 */
#define VRSAVE_COND 7                   /* recommended VR condition bit */

#undef VOLATILE_r13                     /* r13 volatile or non-volatile */

#define MIN_STACK_ALIGN  16
```

salppc.inc 3/9/2001

```
#define MIN_STACK_ALIGN_MASK  (MIN_STACK_ALIGN - 1)

#define ALIGN STACK( nbytes ) \
  (((nbytes) + MIN_STACK_ALIGN_MASK) & ~MIN_STACK_ALIGN_MASK)

#define LR SAVE OFF  4
#define FPR_SAVE_OFF  (-(32-14)*8)

#if defined( VOLATILE_r13 )
#define GPR_SAVE_OFF  (FPR_SAVE_OFF - (32-14)*4)
#else
#define GPR_SAVE_OFF  (FPR_SAVE_OFF - (32-13)*4)
#endif

#define CR_SAVE_OFF  (GPR_SAVE_OFF - 4)

#if defined( BUILD_MAX )

#define VRSAVE_SAVE_OFF  (CR_SAVE_OFF - 4)

#if defined( VOLATILE r13 )
#define ALIGNMENT_PADDING_OFF  (VRSAVE_SAVE_OFF - 0)
#else
#define ALIGNMENT_PADDING_OFF  (VRSAVE_SAVE_OFF - 12)
#endif

#define VR SAVE OFF  (ALIGNMENT_PADDING_OFF - (32-20)*16)
#define LAST_OFF  VR_SAVE_OFF

#else

#define LAST_OFF  CR_SAVE_OFF

#endif

#define REG SAVE SIZE  (-LAST_OFF)
#define MAX NARGS  18
#define ARGS SIZE  (MAX_NARGS * 4)
#define LINK SIZE  8
#define STACK_FRAME_SIZE  (REG_SAVE_SIZE + ARGS_SIZE + LINK_SIZE)

/*
 *  macros to obtain the byte offset into the stack for the last FPR
 *  and GPR registers for small temporary storage.
 *  FPR_SAVE AREA OFFSET points to an area of 8 * (# of unsaved non-volatile
 *  FPR registers).
 *  GPR_SAVE AREA OFFSET points to an area of 4 * (# of unsaved non-volatile
 *  GPR registers).
 *  GET FPR SAVE AREA places the start of the FPR save area into a register
 *  GET_GPR_SAVE_AREA places the start of the GPR save area into a register
 *
 *  For MAX only:
 *
 *  VR_SAVE AREA OFFSET points to an area of 16 * (# of unsaved non-volatile
 *  VR registers).
 *  GET_VR_SAVE_AREA places the start of the VR save area into a register
 */
#define FPR SAVE AREA OFFSET FPR SAVE OFF
#define GPR_SAVE_AREA_OFFSET GPR_SAVE_OFF

#define GET FPR SAVE AREA( ptr ) \
   addi ptr, sp, FPR_SAVE_AREA_OFFSET;

#define GET GPR SAVE AREA( ptr ) \
   addi ptr, sp, GPR_SAVE_AREA_OFFSET;

#if defined( BUILD_MAX )
```

salppc.inc 3/9/2001

```
#define VR_SAVE_AREA_OFFSET VR_SAVE_OFF

#define GET_VR_SAVE_AREA( ptr ) \
   addi ptr, sp, VR_SAVE_AREA_OFFSET;
#endif

/*
 *  if the function creates a stack frame with local storage,
 *  LOCAL_STORAGE_OFFSET is the stack offset to the start of this
 *  storage and is guaranteed to have the minimum stack alignment.
 */
#define LOCAL_STORAGE_OFFSET  (LINK_SIZE + ARGS_SIZE)

/*
 * macros to create and destroy a stack frame.
 *
 * CREATE_STACK_FRAME[_X] creates a stack frame that can handle up to
 * 18 GPR register arguments and a local storage size <=
 * 32768 - 512 = 32,256 bytes.
 *
 * CREATE_STACK_FRAME_X destroys r0.
 *
 * For CREATE_STACK_FRAME_X, local_nbytes_reg must not be r0.
 *
 * Both CREATE_STACK_FRAME[_X] and DESTROY_STACK_FRAME should not be
 * called before registers are saved or after they are restored.
 *
 * The stack pointer "output from" CREATE_STACK_FRAME[_X] must be
 * the same "input to" DESTROY_STACK_FRAME.
 */
#define CREATE_STACK_FRAME( local_nbytes ) \
   stwu sp, -ALIGN_STACK( STACK_FRAME_SIZE + (local_nbytes) )(sp);

#define CREATE_STACK_FRAME_X( local_nbytes_reg ) \
   addi r0, local_nytes_reg, (STACK_FRAME_SIZE + MIN_STACK_ALIGN_SIZE); \
   andi. r0, r0, ~MIN_STACK_ALIGN_MASK; \
   stwux sp, sp, r0;

#define DESTROY_STACK_FRAME \
   lwz sp, 0(sp);

/*
 *  macros to allocate and free space on the user stack.
 *  with a fixed alignment of MIN_STACK_ALIGN.
 *  nbytes must be <= (32768 - 432 = 32,336).
 *  On return, sp points to a buffer of nbytes bytes.
 */
#define PUSH_STACK( nbytes ) \
   addi sp, sp, -ALIGN_STACK( REG_SAVE_SIZE + (nbytes) );

#define POP_STACK( nbytes ) \
   addi sp, sp, ALIGN_STACK( REG_SAVE_SIZE + (nbytes) );

#define ALLOCATE_STACK_SPACE( ptr, nbytes ) \
   PUSH_STACK( nbytes ) \
   mr ptr, sp;

#define FREE_STACK_SPACE( nbytes ) POP_STACK( nbytes )

/*
 * macros to create and destroy a stack buffer with a variable
 * alignment and size.
 *
 * CREATE_STACK_BUFFER[_X] creates a buffer of size nbytes and alignment
 * byte_align on the stack, returning a pointer to the buffer in the
 * GPR bufferp.
 *
```

```
 * bufferp must be a GPR other than r0 and r1 (sp).
 * byte align must be a power of 2 such that 2 <= byte_align <= 4096.
 * CREATE_STACK_BUFFER destroys r0.
 *
 * CREATE STACK BUFFER[ X] stores the original value of the stack pointer
 * below the buffer at offset 0 from the new stack pointer.
 *
 * DESTROY STACK BUFFER sets the stack pointer to the value stored
 * at the address pointed to by the input stack pointer.
 *
 * Both CREATE STACK BUFFER[ X] and DESTROY STACK BUFFER should not be
 * called before registers are saved or after they are restored.
 *
 * The stack pointer "output from" CREATE STACK_BUFFER[_X] must be
 * the same "input to" DESTROY_STACK_BUFFER.
 */
#define CREATE STACK BUFFER( bufferp, byte align, nbytes ) \
   addis bufferp, sp, (-(REG SAVE SIZE + (nbytes)) + 32768)@h; \
   li r0, (((byte align) - 1) | MIN STACK ALIGN MASK); \
   addi bufferp, bufferp, (-(REG_SAVE_SIZE + (nbytes)))@l; \
   andc bufferp, bufferp, r0; \
   sub r0, bufferp, sp; \
   addic r0, r0, -MIN_STACK_ALIGN; \
   stwux sp, sp, r0;

#define CREATE STACK BUFFER X( bufferp, byte_align, nbytes_reg') \
   sub bufferp, sp, nbytes_reg; \
   li r0, (((byte align) - 1) | MIN STACK_ALIGN_MASK); \
   addi bufferp, bufferp, -REG SAVE_SIZE; \
   andc bufferp, bufferp, r0; \
   sub r0, bufferp, sp; \
   addic r0, r0, -MIN_STACK_ALIGN; \
   stwux sp, sp, r0;

#define DESTROY STACK_BUFFER \
   lwz sp, 0(sp);

/*
 * macros to create and destroy the salcache buffer on the user stack.
 *
 * CREATE_STACK_SALCACHE destroys r0.
 *
 * Both CREATE STACK SALCACHE and DESTROY STACK SALCACHE should not be
 * called before registers are saved or after they are restored.
 */
#define CREATE STACK SALCACHE( cachep ) \
        CREATE_STACK_BUFFER( cachep, SALCACHE_ALIGN, SALCACHE_ALLOC_SIZE )

#define DESTROY_STACK_SALCACHE DESTROY_STACK_BUFFER

/*
 *  macros for saving and restoring non-volatile
 *  floating point registers (FPRs)
 */
#define SAVE f14  SR_f14( stfd )
#define SAVE f14 f15  SR f14 f15( stfd )
#define SAVE f14 f16  SR f14 f16( stfd )
#define SAVE f14 f17  SR f14 f17( stfd )
#define SAVE f14 f18  SR f14 f18( stfd )
#define SAVE f14 f19  SR f14 f19( stfd )
#define SAVE f14 f20  SR f14 f20( stfd )
#define SAVE f14 f21  SR f14 f21( stfd )
#define SAVE f14 f22  SR f14 f22( stfd )
#define SAVE f14 f23  SR f14 f23( stfd )
#define SAVE f14 f24  SR f14 f24( stfd )
#define SAVE f14 f25  SR f14 f25( stfd )
#define SAVE_f14_f26  SR_f14_f26( stfd )
```

salppc.inc 3/9/2001

```
#define SAVE f14 f27  SR f14 f27( stfd )
#define SAVE f14 f28  SR f14 f28( stfd )
#define SAVE f14 f29  SR f14 f29( stfd )
#define SAVE f14 f30  SR f14 f30( stfd )
#define SAVE_f14_f31  SR_f14_f31( stfd )

#define SAVE d14  SR_f14( stfd )
#define SAVE d14 d15  SR f14 f15( stfd )
#define SAVE d14 d16  SR f14 f16( stfd )
#define SAVE d14 d17  SR f14 f17( stfd )
#define SAVE d14 d18  SR f14 f18( stfd )
#define SAVE d14 d19  SR f14 f19( stfd )
#define SAVE d14 d20  SR f14 f20( stfd )
#define SAVE d14 d21  SR f14 f21( stfd )
#define SAVE d14 d22  SR f14 f22( stfd )
#define SAVE d14 d23  SR f14 f23( stfd )
#define SAVE d14 d24  SR f14 f24( stfd )
#define SAVE d14 d25  SR f14 f25( stfd )
#define SAVE d14 d26  SR f14 f26( stfd )
#define SAVE d14 d27  SR f14 f27( stfd )
#define SAVE d14 d28  SR f14 f28( stfd )
#define SAVE d14 d29  SR f14 f29( stfd )
#define SAVE d14 d30  SR f14 f30( stfd )
#define SAVE_d14_d31  SR_f14_f31( stfd )

#define REST f14  SR_f14( lfd )
#define REST f14 f15  SR f14 f15( lfd )
#define REST f14 f16  SR f14 f16( lfd )
#define REST f14 f17  SR f14 f17( lfd )
#define REST f14 f18  SR f14 f18( lfd )
#define REST f14 f19  SR f14 f19( lfd )
#define REST f14 f20  SR f14 f20( lfd )
#define REST f14 f21  SR f14 f21( lfd )
#define REST f14 f22  SR f14 f22( lfd )
#define REST f14 f23  SR f14 f23( lfd )
#define REST f14 f24  SR f14 f24( lfd )
#define REST f14 f25  SR f14 f25( lfd )
#define REST f14 f26  SR f14 f26( lfd )
#define REST f14 f27  SR f14 f27( lfd )
#define REST f14 f28  SR f14 f28( lfd )
#define REST f14 f29  SR f14 f29( lfd )
#define REST f14 f30  SR f14 f30( lfd )
#define REST_f14_f31  SR_f14_f31( lfd )

#define REST d14  SR_f14( lfd )
#define REST d14 d15  SR f14 f15( lfd )
#define REST d14 d16  SR f14 f16( lfd )
#define REST d14 d17  SR f14 f17( lfd )
#define REST d14 d18  SR f14 f18( lfd )
#define REST d14 d19  SR f14 f19( lfd )
#define REST d14 d20  SR f14 f20( lfd )
#define REST d14 d21  SR f14 f21( lfd )
#define REST d14 d22  SR f14 f22( lfd )
#define REST d14 d23  SR f14 f23( lfd )
#define REST d14 d24  SR f14 f24( lfd )
#define REST d14 d25  SR f14 f25( lfd )
#define REST d14 d26  SR f14 f26( lfd )
#define REST d14 d27  SR f14 f27( lfd )
#define REST d14 d28  SR f14 f28( lfd )
#define REST d14 d29  SR f14 f29( lfd )
#define REST d14 d30  SR f14 f30( lfd )
#define REST_d14_d31  SR_f14_f31( lfd )

/*
 *  macros common to both FPR save and restore
 */
#define SR_f14( opcode ) \
```

```
   opcode f14, (FPR_SAVE OFF + 17*8)(sp);
#define SR f14_f15( opcode ) \
   opcode f15, (FPR_SAVE_OFF + 16*8)(sp); \
   SR f14( opcode )
#define SR f14_f16( opcode ) \
   opcode f16, (FPR SAVE_OFF + 15*8)(sp); \
   SR f14 f15( opcode )
#define SR f14_f17( opcode ) \
   opcode f17, (FPR SAVE_OFF + 14*8)(sp); \
   SR f14 f16( opcode )
#define SR f14_f18( opcode ) \
   opcode f18, (FPR SAVE_OFF + 13*8)(sp); \
   SR f14 f17( opcode )
#define SR f14_f19( opcode ) \
   opcode f19, (FPR SAVE_OFF + 12*8)(sp); \
   SR f14 f18( opcode )
#define SR f14_f20( opcode ) \
   opcode f20, (FPR SAVE_OFF + 11*8)(sp); \
   SR f14 f19( opcode )
#define SR f14_f21( opcode ) \
   opcode f21, (FPR SAVE_OFF + 10*8)(sp); \
   SR f14 f20( opcode )
#define SR f14_f22( opcode ) \
   opcode f22, (FPR SAVE_OFF + 9*8)(sp); \
   SR f14 f21( opcode )
#define SR f14_f23( opcode ) \
   opcode f23, (FPR SAVE_OFF + 8*8)(sp); \
   SR f14 f22( opcode )
#define SR f14_f24( opcode ) \
   opcode f24, (FPR SAVE_OFF + 7*8)(sp); \
   SR f14 f23( opcode )
#define SR f14_f25( opcode ) \
   opcode f25, (FPR SAVE_OFF + 6*8)(sp); \
   SR f14 f24( opcode )
#define SR f14_f26( opcode ) \
   opcode f26, (FPR SAVE_OFF + 5*8)(sp); \
   SR f14 f25( opcode )
#define SR f14_f27( opcode ) \
   opcode f27, (FPR SAVE_OFF + 4*8)(sp); \
   SR f14 f26( opcode )
#define SR f14_f28( opcode ) \
   opcode f28, (FPR SAVE_OFF + 3*8)(sp); \
   SR f14 f27( opcode )
#define SR f14_f29( opcode ) \
   opcode f29, (FPR SAVE_OFF + 2*8)(sp); \
   SR f14 f28( opcode )
#define SR f14_f30( opcode ) \
   opcode f30, (FPR SAVE_OFF + 1*8)(sp); \
   SR f14 f29( opcode )
#define SR f14_f31( opcode ) \
   opcode f31, (FPR SAVE_OFF)(sp); \
   SR_f14_f30( opcode )

/*
 *  macros for saving and restoring non-volatile
 *  general purpose registers (GPRs)
 */
#if defined( VOLATILE_r13 )

#define SAVE r13
#define SAVE r13 r14   SR r14( stw )
#define SAVE r13 r15   SR r14 r15( stw )
#define SAVE r13 r16   SR r14 r16( stw )
#define SAVE r13 r17   SR r14 r17( stw )
#define SAVE r13 r18   SR r14 r18( stw )
#define SAVE r13 r19   SR r14 r19( stw )
#define SAVE_r13_r20   SR_r14_r20( stw )
```

salppc.inc 3/9/2001

```
#define SAVE r13 r21  SR r14 r21( stw )
#define SAVE r13 r22  SR r14 r22( stw )
#define SAVE r13 r23  SR r14 r23( stw )
#define SAVE r13 r24  SR r14 r24( stw )
#define SAVE r13 r25  SR r14 r25( stw )
#define SAVE r13 r26  SR r14 r26( stw )
#define SAVE r13 r27  SR r14 r27( stw )
#define SAVE r13 r28  SR r14 r28( stw )
#define SAVE r13 r29  SR r14 r29( stw )
#define SAVE r13 r30  SR r14 r30( stw )
#define SAVE_r13_r31  SR_r14_r31( stw )

#define REST r13
#define REST r13 r14  SR r14( lwz )
#define REST r13 r15  SR r14 r15( lwz )
#define REST r13 r16  SR r14 r16( lwz )
#define REST r13 r17  SR r14 r17( lwz )
#define REST r13 r18  SR r14 r18( lwz )
#define REST r13 r19  SR r14 r19( lwz )
#define REST r13 r20  SR r14 r20( lwz )
#define REST r13 r21  SR r14 r21( lwz )
#define REST r13 r22  SR r14 r22( lwz )
#define REST r13 r23  SR r14 r23( lwz )
#define REST r13 r24  SR r14 r24( lwz )
#define REST r13 r25  SR r14 r25( lwz )
#define REST r13 r26  SR r14 r26( lwz )
#define REST r13 r27  SR r14 r27( lwz )
#define REST r13 r28  SR r14 r28( lwz )
#define REST r13 r29  SR r14 r29( lwz )
#define REST r13 r30  SR r14 r30( lwz )
#define REST_r13_r31  SR_r14_r31( lwz )

#else                                 /* r13 is non-volatile */

#define SAVE r13  SR_r13( stw )
#define SAVE r13 r14  SR r13 r14( stw )
#define SAVE r13 r15  SR r13 r15( stw )
#define SAVE r13 r16  SR r13 r16( stw )
#define SAVE r13 r17  SR r13 r17( stw )
#define SAVE r13 r18  SR r13 r18( stw )
#define SAVE r13 r19  SR r13 r19( stw )
#define SAVE r13 r20  SR r13 r20( stw )
#define SAVE r13 r21  SR r13 r21( stw )
#define SAVE r13 r22  SR r13 r22( stw )
#define SAVE r13 r23  SR r13 r23( stw )
#define SAVE r13 r24  SR r13 r24( stw )
#define SAVE r13 r25  SR r13 r25( stw )
#define SAVE r13 r26  SR r13 r26( stw )
#define SAVE r13 r27  SR r13 r27( stw )
#define SAVE r13 r28  SR r13 r28( stw )
#define SAVE r13 r29  SR r13 r29( stw )
#define SAVE r13 r30  SR r13 r30( stw )
#define SAVE_r13_r31  SR_r13_r31( stw )

#define REST r13  SR_r13( lwz )
#define REST r13 r14  SR r13 r14( lwz )
#define REST r13 r15  SR r13 r15( lwz )
#define REST r13 r16  SR r13 r16( lwz )
#define REST r13 r17  SR r13 r17( lwz )
#define REST r13 r18  SR r13 r18( lwz )
#define REST r13 r19  SR r13 r19( lwz )
#define REST r13 r20  SR r13 r20( lwz )
#define REST r13 r21  SR r13 r21( lwz )
#define REST r13 r22  SR r13 r22( lwz )
#define REST r13 r23  SR r13 r23( lwz )
#define REST r13 r24  SR r13 r24( lwz )
#define REST_r13_r25  SR_r13_r25( lwz )
```

salppc.inc 3/9/2001

```
#define REST_r13_r26  SR_r13_r26( lwz )
#define REST_r13_r27  SR_r13_r27( lwz )
#define REST_r13_r28  SR_r13_r28( lwz )
#define REST_r13_r29  SR_r13_r29( lwz )
#define REST_r13_r30  SR_r13_r30( lwz )
#define REST_r13_r31  SR_r13_r31( lwz )

/*
 *  macros common to both GPR save and restore
 */
#define SR_r13( opcode ) \
   opcode r13, (GPR_SAVE_OFF + 18*4)(sp);
#define SR_r13_r14( opcode ) \
   opcode r14, (GPR_SAVE_OFF + 17*4)(sp); \
   SR_r13( opcode )
#define SR_r13_r15( opcode ) \
   opcode r15, (GPR_SAVE_OFF + 16*4)(sp); \
   SR_r13_r14( opcode )
#define SR_r13_r16( opcode ) \
   opcode r16, (GPR_SAVE_OFF + 15*4)(sp); \
   SR_r13_r15( opcode )
#define SR_r13_r17( opcode ) \
   opcode r17, (GPR_SAVE_OFF + 14*4)(sp); \
   SR_r13_r16( opcode )
#define SR_r13_r18( opcode ) \
   opcode r18, (GPR_SAVE_OFF + 13*4)(sp); \
   SR_r13_r17( opcode )
#define SR_r13_r19( opcode ) \
   opcode r19, (GPR_SAVE_OFF + 12*4)(sp); \
   SR_r13_r18( opcode )
#define SR_r13_r20( opcode ) \
   opcode r20, (GPR_SAVE_OFF + 11*4)(sp); \
   SR_r13_r19( opcode )
#define SR_r13_r21( opcode ) \
   opcode r21, (GPR_SAVE_OFF + 10*4)(sp); \
   SR_r13_r20( opcode )
#define SR_r13_r22( opcode ) \
   opcode r22, (GPR_SAVE_OFF + 9*4)(sp); \
   SR_r13_r21( opcode )
#define SR_r13_r23( opcode ) \
   opcode r23, (GPR_SAVE_OFF + 8*4)(sp); \
   SR_r13_r22( opcode )
#define SR_r13_r24( opcode ) \
   opcode r24, (GPR_SAVE_OFF + 7*4)(sp); \
   SR_r13_r23( opcode )
#define SR_r13_r25( opcode ) \
   opcode r25, (GPR_SAVE_OFF + 6*4)(sp); \
   SR_r13_r24( opcode )
#define SR_r13_r26( opcode ) \
   opcode r26, (GPR_SAVE_OFF + 5*4)(sp); \
   SR_r13_r25( opcode )
#define SR_r13_r27( opcode ) \
   opcode r27, (GPR_SAVE_OFF + 4*4)(sp); \
   SR_r13_r26( opcode )
#define SR_r13_r28( opcode ) \
   opcode r28, (GPR_SAVE_OFF + 3*4)(sp); \
   SR_r13_r27( opcode )
#define SR_r13_r29( opcode ) \
   opcode r29, (GPR_SAVE_OFF + 2*4)(sp); \
   SR_r13_r28( opcode )
#define SR_r13_r30( opcode ) \
   opcode r30, (GPR_SAVE_OFF + 1*4)(sp); \
   SR_r13_r29( opcode )
#define SR_r13_r31( opcode ) \
   opcode r31, (GPR_SAVE_OFF)(sp); \
   SR_r13_r30( opcode )
```

```
#endif                                  /* end VOLATILE_r13 */

#define SAVE_r14   SR_r14( stw )
#define SAVE_r14_r15  SR_r14_r15( stw )
#define SAVE_r14_r16  SR_r14_r16( stw )
#define SAVE_r14_r17  SR_r14_r17( stw )
#define SAVE_r14_r18  SR_r14_r18( stw )
#define SAVE_r14_r19  SR_r14_r19( stw )
#define SAVE_r14_r20  SR_r14_r20( stw )
#define SAVE_r14_r21  SR_r14_r21( stw )
#define SAVE_r14_r22  SR_r14_r22( stw )
#define SAVE_r14_r23  SR_r14_r23( stw )
#define SAVE_r14_r24  SR_r14_r24( stw )
#define SAVE_r14_r25  SR_r14_r25( stw )
#define SAVE_r14_r26  SR_r14_r26( stw )
#define SAVE_r14_r27  SR_r14_r27( stw )
#define SAVE_r14_r28  SR_r14_r28( stw )
#define SAVE_r14_r29  SR_r14_r29( stw )
#define SAVE_r14_r30  SR_r14_r30( stw )
#define SAVE_r14_r31  SR_r14_r31( stw )

#define REST_r14   SR_r14( lwz )
#define REST_r14_r15  SR_r14_r15( lwz )
#define REST_r14_r16  SR_r14_r16( lwz )
#define REST_r14_r17  SR_r14_r17( lwz )
#define REST_r14_r18  SR_r14_r18( lwz )
#define REST_r14_r19  SR_r14_r19( lwz )
#define REST_r14_r20  SR_r14_r20( lwz )
#define REST_r14_r21  SR_r14_r21( lwz )
#define REST_r14_r22  SR_r14_r22( lwz )
#define REST_r14_r23  SR_r14_r23( lwz )
#define REST_r14_r24  SR_r14_r24( lwz )
#define REST_r14_r25  SR_r14_r25( lwz )
#define REST_r14_r26  SR_r14_r26( lwz )
#define REST_r14_r27  SR_r14_r27( lwz )
#define REST_r14_r28  SR_r14_r28( lwz )
#define REST_r14_r29  SR_r14_r29( lwz )
#define REST_r14_r30  SR_r14_r30( lwz )
#define REST_r14_r31  SR_r14_r31( lwz )

/*
 *  macros common to both GPR save and restore
 */
#define SR_r14( opcode ) \
   opcode r14, (GPR_SAVE_OFF + 17*4)(sp);
#define SR_r14_r15( opcode ) \
   opcode r15, (GPR_SAVE_OFF + 16*4)(sp); \
   SR_r14( opcode )
#define SR_r14_r16( opcode ) \
   opcode r16, (GPR_SAVE_OFF + 15*4)(sp); \
   SR_r14_r15( opcode )
#define SR_r14_r17( opcode ) \
   opcode r17, (GPR_SAVE_OFF + 14*4)(sp); \
   SR_r14_r16( opcode )
#define SR_r14_r18( opcode ) \
   opcode r18, (GPR_SAVE_OFF + 13*4)(sp); \
   SR_r14_r17( opcode )
#define SR_r14_r19( opcode ) \
   opcode r19, (GPR_SAVE_OFF + 12*4)(sp); \
   SR_r14_r18( opcode )
#define SR_r14_r20( opcode ) \
   opcode r20, (GPR_SAVE_OFF + 11*4)(sp); \
   SR_r14_r19( opcode )
#define SR_r14_r21( opcode ) \
   opcode r21, (GPR_SAVE_OFF + 10*4)(sp); \
   SR_r14_r20( opcode )
#define SR_r14_r22( opcode ) \
```

salppc.inc 3/9/2001

```
    opcode r22, (GPR SAVE_OFF + 9*4)(sp); \
    SR r14 r21( opcode )
#define SR r14_r23( opcode ) \
    opcode r23, (GPR SAVE_OFF + 8*4)(sp); \
    SR r14 r22( opcode )
#define SR r14_r24( opcode ) \
    opcode r24, (GPR SAVE_OFF + 7*4)(sp); \
    SR r14 r23( opcode )
#define SR r14_r25( opcode ) \
    opcode r25, (GPR SAVE_OFF + 6*4)(sp); \
    SR r14 r24( opcode )
#define SR r14_r26( opcode ) \
    opcode r26, (GPR SAVE_OFF + 5*4)(sp); \
    SR r14 r25( opcode )
#define SR r14_r27( opcode ) \
    opcode r27, (GPR SAVE_OFF + 4*4)(sp); \
    SR r14 r26( opcode )
#define SR r14_r28( opcode ) \
    opcode r28, (GPR SAVE_OFF + 3*4)(sp); \
    SR r14 r27( opcode )
#define SR r14_r29( opcode ) \
    opcode r29, (GPR SAVE_OFF + 2*4)(sp); \
    SR r14 r28( opcode )
#define SR r14_r30( opcode ) \
    opcode r30, (GPR SAVE_OFF + 1*4)(sp); \
    SR r14 r29( opcode )
#define SR r14_r31( opcode ) \
    opcode r31, (GPR SAVE_OFF)(sp); \
    SR_r14_r30( opcode )

#define SAVE r15   SR_r15( stw )
#define SAVE r15 r16   SR r15 r16( stw )
#define SAVE r15 r17   SR r15 r17( stw )
#define SAVE r15 r18   SR r15 r18( stw )
#define SAVE r15 r19   SR r15 r19( stw )
#define SAVE r15 r20   SR r15 r20( stw )
#define SAVE r15 r21   SR r15 r21( stw )
#define SAVE r15 r22   SR r15 r22( stw )
#define SAVE r15 r23   SR r15 r23( stw )
#define SAVE r15 r24   SR r15 r24( stw )
#define SAVE r15 r25   SR r15 r25( stw )
#define SAVE r15 r26   SR r15 r26( stw )
#define SAVE r15 r27   SR r15 r27( stw )
#define SAVE r15 r28   SR r15 r28( stw )
#define SAVE r15 r29   SR r15 r29( stw )
#define SAVE r15 r30   SR r15 r30( stw )
#define SAVE_r15_r31   SR_r15_r31( stw )

#define REST r15   SR_r15( lwz )
#define REST r15 r16   SR r15 r16( lwz )
#define REST r15 r17   SR r15 r17( lwz )
#define REST r15 r18   SR r15 r18( lwz )
#define REST r15 r19   SR r15 r19( lwz )
#define REST r15 r20   SR r15 r20( lwz )
#define REST r15 r21   SR r15 r21( lwz )
#define REST r15 r22   SR r15 r22( lwz )
#define REST r15 r23   SR r15 r23( lwz )
#define REST r15 r24   SR r15 r24( lwz )
#define REST r15 r25   SR r15 r25( lwz )
#define REST r15 r26   SR r15 r26( lwz )
#define REST r15 r27   SR r15 r27( lwz )
#define REST r15 r28   SR r15 r28( lwz )
#define REST r15 r29   SR r15 r29( lwz )
#define REST r15 r30   SR r15 r30( lwz )
#define REST_r15_r31   SR_r15_r31( lwz )

/*
```

```
 *  macros common to both GPR save and restore
 */
#define SR r15( opcode ) \
   opcode r15, (GPR_SAVE OFF + 16*4)(sp);
#define SR r15_r16( opcode ) \
   opcode r16, (GPR_SAVE_OFF + 15*4)(sp); \
   SR r15( opcode )
#define SR r15_r17( opcode ) \
   opcode r17, (GPR SAVE_OFF + 14*4)(sp); \
   SR r15 r16( opcode )
#define SR r15_r18( opcode ) \
   opcode r18, (GPR SAVE_OFF + 13*4)(sp); \
   SR r15 r17( opcode )
#define SR r15_r19( opcode ) \
   opcode r19, (GPR SAVE_OFF + 12*4)(sp); \
   SR r15 r18( opcode )
#define SR r15_r20( opcode ) \
   opcode r20, (GPR SAVE_OFF + 11*4)(sp); \
   SR r15 r19( opcode )
#define SR r15_r21( opcode ) \
   opcode r21, (GPR SAVE_OFF + 10*4)(sp); \
   SR r15 r20( opcode )
#define SR r15_r22( opcode ) \
   opcode r22, (GPR SAVE_OFF + 9*4)(sp); \
   SR r15 r21( opcode )
#define SR r15_r23( opcode ) \
   opcode r23, (GPR SAVE_OFF + 8*4)(sp); \
   SR r15 r22( opcode )
#define SR r15_r24( opcode ) \
   opcode r24, (GPR SAVE_OFF + 7*4)(sp); \
   SR r15 r23( opcode )
#define SR r15_r25( opcode ) \
   opcode r25, (GPR SAVE_OFF + 6*4)(sp); \
   SR r15 r24( opcode )
#define SR r15_r26( opcode ) \
   opcode r26, (GPR SAVE_OFF + 5*4)(sp); \
   SR r15 r25( opcode )
#define SR r15_r27( opcode ) \
   opcode r27, (GPR SAVE_OFF + 4*4)(sp); \
   SR r15 r26( opcode )
#define SR r15_r28( opcode ) \
   opcode r28, (GPR SAVE_OFF + 3*4)(sp); \
   SR r15 r27( opcode )
#define SR r15_r29( opcode ) \
   opcode r29, (GPR SAVE_OFF + 2*4)(sp); \
   SR r15 r28( opcode )
#define SR r15_r30( opcode ) \
   opcode r30, (GPR SAVE_OFF + 1*4)(sp); \
   SR r15 r29( opcode )
#define SR r15_r31( opcode ) \
   opcode r31, (GPR SAVE_OFF)(sp); \
   SR_r15_r30( opcode )

#define SAVE r16  SR_r16( stw )
#define SAVE r16 r17  SR r16 r17( stw )
#define SAVE r16 r18  SR r16 r18( stw )
#define SAVE r16 r19  SR r16 r19( stw )
#define SAVE r16 r20  SR r16 r20( stw )
#define SAVE r16 r21  SR r16 r21( stw )
#define SAVE r16 r22  SR r16 r22( stw )
#define SAVE r16 r23  SR r16 r23( stw )
#define SAVE r16 r24  SR r16 r24( stw )
#define SAVE r16 r25  SR r16 r25( stw )
#define SAVE r16 r26  SR r16 r26( stw )
#define SAVE r16 r27  SR r16 r27( stw )
#define SAVE r16 r28  SR r16 r28( stw )
#define SAVE_r16_r29  SR_r16_r29( stw )
```

salppc.inc 3/9/2001

```
#define SAVE r16 r30  SR r16 r30( stw )
#define SAVE_r16_r31  SR_r16_r31( stw )

#define REST r16  SR_r16( lwz )
#define REST r16 r17  SR r16 r17( lwz )
#define REST r16 r18  SR r16 r18( lwz )
#define REST r16 r19  SR r16 r19( lwz )
#define REST r16 r20  SR r16 r20( lwz )
#define REST r16 r21  SR r16 r21( lwz )
#define REST r16 r22  SR r16 r22( lwz )
#define REST r16 r23  SR r16 r23( lwz )
#define REST r16 r24  SR r16 r24( lwz )
#define REST r16 r25  SR r16 r25( lwz )
#define REST r16 r26  SR r16 r26( lwz )
#define REST r16 r27  SR r16 r27( lwz )
#define REST r16 r28  SR r16 r28( lwz )
#define REST r16 r29  SR r16 r29( lwz )
#define REST r16 r30  SR r16 r30( lwz )
#define REST_r16_r31  SR_r16_r31( lwz )

/*
 *  macros common to both GPR save and restore
 */
#define SR r16( opcode ) \
   opcode r16, (GPR_SAVE OFF + 15*4)(sp);
#define SR r16_r17( opcode ) \
   opcode r17, (GPR_SAVE_OFF + 14*4)(sp); \
   SR r16( opcode )
#define SR r16_r18( opcode ) \
   opcode r18, (GPR SAVE_OFF + 13*4)(sp); \
   SR r16 r17( opcode )
#define SR r16_r19( opcode ) \
   opcode r19, (GPR SAVE_OFF + 12*4)(sp); \
   SR r16 r18( opcode )
#define SR r16_r20( opcode ) \
   opcode r20, (GPR SAVE_OFF + 11*4)(sp); \
   SR r16 r19( opcode )
#define SR r16_r21( opcode ) \
   opcode r21, (GPR SAVE_OFF + 10*4)(sp); \
   SR r16 r20( opcode )
#define SR r16_r22( opcode ) \
   opcode r22, (GPR SAVE_OFF + 9*4)(sp); \
   SR r16 r21( opcode )
#define SR r16_r23( opcode ) \
   opcode r23, (GPR SAVE_OFF + 8*4)(sp); \
   SR r16 r22( opcode )
#define SR r16_r24( opcode ) \
   opcode r24, (GPR SAVE_OFF + 7*4)(sp); \
   SR r16 r23( opcode )
#define SR r16_r25( opcode ) \
   opcode r25, (GPR SAVE_OFF + 6*4)(sp); \
   SR r16 r24( opcode )
#define SR r16_r26( opcode ) \
   opcode r26, (GPR SAVE_OFF + 5*4)(sp); \
   SR r16 r25( opcode )
#define SR r16_r27( opcode ) \
   opcode r27, (GPR SAVE_OFF + 4*4)(sp); \
   SR r16 r26( opcode )
#define SR r16_r28( opcode ) \
   opcode r28, (GPR SAVE_OFF + 3*4)(sp); \
   SR r16 r27( opcode )
#define SR r16_r29( opcode ) \
   opcode r29, (GPR SAVE_OFF + 2*4)(sp); \
   SR r16 r28( opcode )
#define SR r16_r30( opcode ) \
   opcode r30, (GPR SAVE_OFF + 1*4)(sp); \
   SR_r16_r29( opcode )
```

```
#define SR r16_r31( opcode ) \
   opcode r31, (GPR SAVE_OFF)(sp); \
   SR_r16_r30( opcode )

#if defined( BUILD_MAX )

/*
 *  macros for saving and restoring non-volatile
 *  vector registers (VRs)
 *  (uses r0 as scratch register)
 */
#define SAVE v20  SR_v20( stvx )
#define SAVE v20 v21  SR v20 v21( stvx )
#define SAVE v20 v22  SR v20 v22( stvx )
#define SAVE v20 v23  SR v20 v23( stvx )
#define SAVE v20 v24  SR v20 v24( stvx )
#define SAVE v20 v25  SR v20 v25( stvx )
#define SAVE v20 v26  SR v20 v26( stvx )
#define SAVE v20 v27  SR v20 v27( stvx )
#define SAVE v20 v28  SR v20 v28( stvx )
#define SAVE v20 v29  SR v20 v29( stvx )
#define SAVE v20 v30  SR v20 v30( stvx )
#define SAVE_v20_v31  SR_v20_v31( stvx )

#define REST v20  SR_v20( lvx )
#define REST v20 v21  SR v20 v21( lvx )
#define REST v20 v22  SR v20 v22( lvx )
#define REST v20 v23  SR v20 v23( lvx )
#define REST v20 v24  SR v20 v24( lvx )
#define REST v20 v25  SR v20 v25( lvx )
#define REST v20 v26  SR v20 v26( lvx )
#define REST v20 v27  SR v20 v27( lvx )
#define REST v20 v28  SR v20 v28( lvx )
#define REST v20 v29  SR v20 v29( lvx )
#define REST v20 v30  SR v20 v30( lvx )
#define REST_v20_v31  SR_v20_v31( lvx )

/*
 *  macros common to both VR save and restore
 *  (uses r0 as scratch register)
 */
#define SR v20( opcode ) \
   li r0, (VR SAVE_OFF + 11*16); \
   opcode v20, sp, r0;
#define SR v20 v21( opcode ) \
   li r0, (VR SAVE_OFF + 10*16); \
   opcode v21, sp, r0; \
   SR v20( opcode )
#define SR v20 v22( opcode ) \
   li r0, (VR SAVE_OFF + 9*16); \
   opcode v22, sp, r0; \
   SR v20 v21( opcode )
#define SR v20 v23( opcode ) \
   li r0, (VR SAVE_OFF + 8*16); \
   opcode v23, sp, r0; \
   SR v20 v22( opcode )
#define SR v20 v24( opcode ) \
   li r0, (VR SAVE_OFF + 7*16); \
   opcode v24, sp, r0; \
   SR v20 v23( opcode )
#define SR v20 v25( opcode ) \
   li r0, (VR SAVE_OFF + 6*16); \
   opcode v25, sp, r0; \
   SR v20 v24( opcode )
#define SR v20 v26( opcode ) \
   li r0, (VR SAVE_OFF + 5*16); \
   opcode v26, sp, r0; \
```

```
   SR_v20_v25( opcode )
#define SR_v20_v27( opcode ) \
   li r0, (VR_SAVE_OFF + 4*16); \
   opcode v27, sp, r0; \
   SR_v20_v26( opcode )
#define SR_v20_v28( opcode ) \
   li r0, (VR_SAVE_OFF + 3*16); \
   opcode v28, sp, r0; \
   SR_v20_v27( opcode )
#define SR_v20_v29( opcode ) \
   li r0, (VR_SAVE_OFF + 2*16); \
   opcode v29, sp, r0; \
   SR_v20_v28( opcode )
#define SR_v20_v30( opcode ) \
   li r0, (VR_SAVE_OFF + 1*16); \
   opcode v30, sp, r0; \
   SR_v20_v29( opcode )
#define SR_v20_v31( opcode ) \
   li r0, (VR_SAVE_OFF); \
   opcode v31, sp, r0; \
   SR_v20_v30( opcode )

/*
 *  macros for saving, updating and restoring VRSAVE and saving and
 *  restoring non-volatile vector registers (v0 - v31)
 *  (destroys r0 and CR0 field of CR)
 */
#define NON_VOLATILE_VR_TEST( last_vreg ) \
   andi. r0, r0, ((-1 << (31 - (last_vreg))) & 0x0fff);

#define RECORD_v0_v15( last_vreg ) \
   oris r0, r0, ((-1 << (15 - (last_vreg))) & 0xffff); \
   mtspr %VRSAVE, r0;

#define RECORD_v16_v31( last_vreg ) \
   oris r0, r0, 0xffff; \
   ori r0, r0, ((-1 << (31 - (last_vreg))) & 0xffff); \
   mtspr %VRSAVE, r0;

#define USE_v0_v15( cond, last_vreg ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET( 8 ); \
   stw r0, VRSAVE_SAVE_OFF(sp); \
   RECORD_v0_v15( last_vreg )

#define USE_v16_v19( cond, last_vreg ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET( 8 ); \
   stw r0, VRSAVE_SAVE_OFF(sp); \
   RECORD_v16_v31( last_vreg )

#define FREE_v0_v19( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET( 8 ); \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

/*
 *  user-callable macros
 */
#define USE_THRU_v0( cond )     USE_v0_v15( cond, 0 )
#define USE_THRU_v1( cond )     USE_v0_v15( cond, 1 )
#define USE_THRU_v2( cond )     USE_v0_v15( cond, 2 )
#define USE_THRU_v3( cond )     USE_v0_v15( cond, 3 )
#define USE_THRU_v4( cond )     USE_v0_v15( cond, 4 )
```

```
#define USE THRU v5( cond )     USE v0 v15( cond, 5 )
#define USE THRU v6( cond )     USE v0 v15( cond, 6 )
#define USE THRU v7( cond )     USE v0 v15( cond, 7 )
#define USE THRU v8( cond )     USE v0 v15( cond, 8 )
#define USE THRU v9( cond )     USE v0 v15( cond, 9 )
#define USE THRU v10( cond )    USE v0 v15( cond, 10 )
#define USE THRU v11( cond )    USE v0 v15( cond, 11 )
#define USE THRU v12( cond )    USE v0 v15( cond, 12 )
#define USE THRU v13( cond )    USE v0 v15( cond, 13 )
#define USE THRU v14( cond )    USE v0 v15( cond, 14 )
#define USE THRU v15( cond )    USE v0 v15( cond, 15 )
#define USE THRU v16( cond )    USE v16 v19( cond, 16 )
#define USE THRU v17( cond )    USE v16 v19( cond, 17 )
#define USE THRU v18( cond )    USE v16 v19( cond, 18 )
#define USE_THRU_v19( cond )    USE_v16_v19( cond, 19 )

#define USE THRU v20( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET( 32 );          /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 20 )            /* v20 in use? */ \
   beq PC_OFFSET(16);                    /* no, cond is set to greater than */
   \
   SAVE v20                              /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;             /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                    /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 20 )                  /* indicate v0 - v20 in use */

#define USE THRU v21( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(40);            /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 21 )            /* v20 - v21 in use? */ \
   beq PC_OFFSET(24);                    /* no, cond is set to greater than */
   \
   SAVE v20 v21                          /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;             /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                    /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 21 )                  /* indicate v0 - v21 in use */

#define USE THRU v22( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(48);            /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 22 )            /* v20 - v22 in use? */ \
   beq PC_OFFSET(32);                    /* no, cond is set to greater than */
   \
   SAVE v20 v22                          /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;             /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                    /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 22 )                  /* indicate v0 - v22 in use */

#define USE THRU v23( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(56);            /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 23 )            /* v20 - v23 in use? */ \
   beq PC_OFFSET(40);                    /* no, cond is set to greater than */
   \
```

```
   SAVE v20 v23                         /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;            /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                   /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 23 )                 /* indicate v0 - v23 in use */

#define USE THRU v24( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(64);           /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 24 )           /* v20 - v24 in use? */ \
   beq PC_OFFSET(48);                   /* no, cond is set to greater than */
   \
   SAVE v20 v24                         /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;            /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                   /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 24 )                 /* indicate v0 - v24 in use */

#define USE THRU v25( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(72);           /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 25 )           /* v20 - v25 in use? */ \
   beq PC_OFFSET(56);                   /* no, cond is set to greater than */
   \
   SAVE v20 v25                         /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;            /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                   /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 25 )                 /* indicate v0 - v25 in use */

#define USE THRU v26( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(80);           /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 26 )           /* v20 - v26 in use? */ \
   beq PC_OFFSET(64);                   /* no, cond is set to greater than */
   \
   SAVE v20 v26                         /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;            /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                   /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 26 )                 /* indicate v0 - v26 in use */

#define USE THRU v27( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(88);           /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE SAVE OFF(sp); \
   NON_VOLATILE VR TEST( 27 )           /* v20 - v27 in use? */ \
   beq PC_OFFSET(72);                   /* no, cond is set to greater than */
   \
   SAVE v20 v27                         /* leaves a negative value in r0 */ \
   cmpwi (cond), r0, 0x7fff;            /* cond is set to less than */ \
   mfspr r0, %VRSAVE;                   /* reload VRSAVE into r0 */ \
   RECORD_v16_v31( 27 )                 /* indicate v0 - v27 in use */

#define USE THRU v28( cond ) \
   mfspr r0, %VRSAVE; \
   cmplwi (cond), r0, 0; \
   beq (cond), PC_OFFSET(96);           /* cond set to equal if VRSAVE = 0 */
   \
   stw r0, VRSAVE_SAVE_OFF(sp); \
```

```
    NON_VOLATILE VR TEST( 28 )          /* v20 - v28 in use? */ \
    beq PC_OFFSET(80);                  /* no, cond is set to greater than */
    \
    SAVE v20 v28                        /* leaves a negative value in r0 */ \
    cmpwi (cond), r0, 0x7fff;           /* cond is set to less than */ \
    mfspr r0, %VRSAVE;                  /* reload VRSAVE into r0 */ \
    RECORD_v16_v31( 28 )                /* indicate v0 - v28 in use */

#define USE THRU v29( cond ) \
    mfspr r0, %VRSAVE; \
    cmplwi (cond), r0, 0; \
    beq (cond), PC_OFFSET(104);         /* cond set to equal if VRSAVE = 0 */
    \
    stw r0, VRSAVE SAVE OFF(sp); \
    NON_VOLATILE VR TEST( 29 )          /* v20 - v29 in use? */ \
    beq PC_OFFSET(88);                  /* no, cond is set to greater than */
    \
    SAVE v20 v29                        /* leaves a negative value in r0 */ \
    cmpwi (cond), r0, 0x7fff;           /* cond is set to less than */ \
    mfspr r0, %VRSAVE;                  /* reload VRSAVE into r0 */ \
    RECORD_v16_v31( 29 )                /* indicate v0 - v29 in use */

#define USE THRU v30( cond ) \
    mfspr r0, %VRSAVE; \
    cmplwi (cond), r0, 0; \
    beq (cond), PC_OFFSET(112);         /* cond set to equal if VRSAVE = 0 */
    \
    stw r0, VRSAVE SAVE OFF(sp); \
    NON_VOLATILE VR TEST( 30 )          /* v20 - v30 in use? */ \
    beq PC_OFFSET(96);                  /* no, cond is set to greater than */
    \
    SAVE v20 v30                        /* leaves a negative value in r0 */ \
    cmpwi (cond), r0, 0x7fff;           /* cond is set to less than */ \
    mfspr r0, %VRSAVE;                  /* reload VRSAVE into r0 */ \
    RECORD_v16_v31( 30 )                /* indicate v0 - v30 in use */

#define USE THRU v31( cond ) \
    mfspr r0, %VRSAVE; \
    cmplwi (cond), r0, 0; \
    beq (cond), PC_OFFSET(120);         /* cond set to equal if VRSAVE = 0 */
    \
    stw r0, VRSAVE SAVE OFF(sp); \
    NON_VOLATILE VR TEST( 31 )          /* v20 - v31 in use? */ \
    beq PC_OFFSET(104);                 /* no, cond is set to greater than */
    \
    SAVE v20 v31                        /* leaves a negative value in r0 */ \
    cmpwi (cond), r0, 0x7fff;           /* cond is set to less than */ \
    mfspr r0, %VRSAVE;                  /* reload VRSAVE into r0 */ \
    RECORD_v16_v31( 31 )                /* indicate v0 - v31 in use */

#define FREE THRU v0( cond )    FREE v0 v19( cond )
#define FREE THRU v1( cond )    FREE v0 v19( cond )
#define FREE THRU v2( cond )    FREE v0 v19( cond )
#define FREE THRU v3( cond )    FREE v0 v19( cond )
#define FREE THRU v4( cond )    FREE v0 v19( cond )
#define FREE THRU v5( cond )    FREE v0 v19( cond )
#define FREE THRU v6( cond )    FREE v0 v19( cond )
#define FREE THRU v7( cond )    FREE v0 v19( cond )
#define FREE THRU v8( cond )    FREE v0 v19( cond )
#define FREE THRU v9( cond )    FREE v0 v19( cond )
#define FREE THRU v10( cond )   FREE v0 v19( cond )
#define FREE THRU v11( cond )   FREE v0 v19( cond )
#define FREE THRU v12( cond )   FREE v0 v19( cond )
#define FREE THRU v13( cond )   FREE v0 v19( cond )
#define FREE THRU v14( cond )   FREE v0 v19( cond )
#define FREE THRU v15( cond )   FREE v0 v19( cond )
#define FREE_THRU_v16( cond )   FREE_v0_v19( cond )
```

```
#define FREE_THRU_v17( cond )  FREE_v0_v19( cond )
#define FREE_THRU_v18( cond )  FREE_v0_v19( cond )
#define FREE_THRU_v19( cond )  FREE_v0_v19( cond )

#define FREE_THRU_v20( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(20); \
   bgt (cond), PC_OFFSET(12); \
   REST_v20; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v21( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(28); \
   bgt (cond), PC_OFFSET(20); \
   REST_v20_v21; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v22( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(36); \
   bgt (cond), PC_OFFSET(28); \
   REST_v20_v22; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v23( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(44); \
   bgt (cond), PC_OFFSET(36); \
   REST_v20_v23; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v24( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(52); \
   bgt (cond), PC_OFFSET(44); \
   REST_v20_v24; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v25( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(60); \
   bgt (cond), PC_OFFSET(52); \
   REST_v20_v25; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v26( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(68); \
   bgt (cond), PC_OFFSET(60); \
   REST_v20_v26; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v27( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(76); \
   bgt (cond), PC_OFFSET(68); \
   REST_v20_v27; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;
```

```
#define FREE_THRU_v28( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(84); \
   bgt (cond), PC_OFFSET(76); \
   REST_v20_v28; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v29( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(92); \
   bgt (cond), PC_OFFSET(84); \
   REST_v20_v29; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v30( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(100); \
   bgt (cond), PC_OFFSET(92); \
   REST_v20_v30; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#define FREE_THRU_v31( cond ) \
   li r0, 0; \
   beq (cond), PC_OFFSET(108); \
   bgt (cond), PC_OFFSET(100); \
   REST_v20_v31; \
   lwz r0, VRSAVE_SAVE_OFF(sp); \
   mtspr %VRSAVE, r0;

#endif                                  /* end BUILD_MAX */

/*
 *  macros to save and restore the CR register
 *  (uses r0 as scratch register)
 */
#define SAVE_CR \
   mfcr r0; \
   stw r0, CR_SAVE_OFF(sp);

#define REST_CR \
   lwz r0, CR_SAVE_OFF(sp); \
   mtcr r0;

/*
 *  macros to save and restore the LR register
 *  (uses r0 as scratch register)
 */
#define SAVE_LR \
   mflr r0; \
   stw r0, LR_SAVE_OFF(sp);

#define REST_LR \
   lwz r0, LR_SAVE_OFF(sp); \
   mtlr r0;

#endif                                  /* end COMPILE_C */

/*
 *  macros for declaring GPR, FPR and VMX registers
 */

/*
 *  declare r0
```

```
 */
#define DECLARE_r0

/*
 *  r3 declare set
 */
#define DECLARE r3
#define DECLARE r3 r4
#define DECLARE r3 r5
#define DECLARE r3 r6
#define DECLARE r3 r7
#define DECLARE r3 r8
#define DECLARE r3 r9
#define DECLARE r3 r10
#define DECLARE r3 r11
#define DECLARE r3 r12
#define DECLARE r3 r13
#define DECLARE r3 r14
#define DECLARE r3 r15
#define DECLARE r3 r16
#define DECLARE r3 r17
#define DECLARE r3 r18
#define DECLARE r3 r19
#define DECLARE r3 r20
#define DECLARE r3 r21
#define DECLARE r3 r22
#define DECLARE r3 r23
#define DECLARE r3 r24
#define DECLARE r3 r25
#define DECLARE r3 r26
#define DECLARE r3 r27
#define DECLARE r3 r28
#define DECLARE r3 r29
#define DECLARE r3 r30
#define DECLARE_r3_r31

/*
 *  r4 declare set
 */
#define DECLARE r4
#define DECLARE r4 r5
#define DECLARE r4 r6
#define DECLARE r4 r7
#define DECLARE r4 r8
#define DECLARE r4 r9
#define DECLARE r4 r10
#define DECLARE r4 r11
#define DECLARE r4 r12
#define DECLARE r4 r13
#define DECLARE r4 r14
#define DECLARE r4 r15
#define DECLARE r4 r16
#define DECLARE r4 r17
#define DECLARE r4 r18
#define DECLARE r4 r19
#define DECLARE r4 r20
#define DECLARE r4 r21
#define DECLARE r4 r22
#define DECLARE r4 r23
#define DECLARE r4 r24
#define DECLARE r4 r25
#define DECLARE r4 r26
#define DECLARE r4 r27
#define DECLARE r4 r28
#define DECLARE r4 r29
#define DECLARE r4 r30
#define DECLARE_r4_r31
```

```
/*
 *  r5 declare set
 */
#define DECLARE r5
#define DECLARE r5 r6
#define DECLARE r5 r7
#define DECLARE r5 r8
#define DECLARE r5 r9
#define DECLARE r5 r10
#define DECLARE r5 r11
#define DECLARE r5 r12
#define DECLARE r5 r13
#define DECLARE r5 r14
#define DECLARE r5 r15
#define DECLARE r5 r16
#define DECLARE r5 r17
#define DECLARE r5 r18
#define DECLARE r5 r19
#define DECLARE r5 r20
#define DECLARE r5 r21
#define DECLARE r5 r22
#define DECLARE r5 r23
#define DECLARE r5 r24
#define DECLARE r5 r25
#define DECLARE r5 r26
#define DECLARE r5 r27
#define DECLARE r5 r28
#define DECLARE r5 r29
#define DECLARE r5 r30
#define DECLARE_r5_r31

/*
 *  r6 declare set
 */
#define DECLARE r6
#define DECLARE r6 r7
#define DECLARE r6 r8
#define DECLARE r6 r9
#define DECLARE r6 r10
#define DECLARE r6 r11
#define DECLARE r6 r12
#define DECLARE r6 r13
#define DECLARE r6 r14
#define DECLARE r6 r15
#define DECLARE r6 r16
#define DECLARE r6 r17
#define DECLARE r6 r18
#define DECLARE r6 r19
#define DECLARE r6 r20
#define DECLARE r6 r21
#define DECLARE r6 r22
#define DECLARE r6 r23
#define DECLARE r6 r24
#define DECLARE r6 r25
#define DECLARE r6 r26
#define DECLARE r6 r27
#define DECLARE r6 r28
#define DECLARE r6 r29
#define DECLARE r6 r30
#define DECLARE_r6_r31

/*
 *  r7 declare set
 */
#define DECLARE r7
#define DECLARE_r7_r8
```

salppc.inc 3/9/2001

```
#define DECLARE r7 r9
#define DECLARE r7 r10
#define DECLARE r7 r11
#define DECLARE r7 r12
#define DECLARE r7 r13
#define DECLARE r7 r14
#define DECLARE r7 r15
#define DECLARE r7 r16
#define DECLARE r7 r17
#define DECLARE r7 r18
#define DECLARE r7 r19
#define DECLARE r7 r20
#define DECLARE r7 r21
#define DECLARE r7 r22
#define DECLARE r7 r23
#define DECLARE r7 r24
#define DECLARE r7 r25
#define DECLARE r7 r26
#define DECLARE r7 r27
#define DECLARE r7 r28
#define DECLARE r7 r29
#define DECLARE r7 r30
#define DECLARE_r7_r31

/*
 *  r8 declare set
 */
#define DECLARE r8
#define DECLARE r8 r9
#define DECLARE r8 r10
#define DECLARE r8 r11
#define DECLARE r8 r12
#define DECLARE r8 r13
#define DECLARE r8 r14
#define DECLARE r8 r15
#define DECLARE r8 r16
#define DECLARE r8 r17
#define DECLARE r8 r18
#define DECLARE r8 r19
#define DECLARE r8 r20
#define DECLARE r8 r21
#define DECLARE r8 r22
#define DECLARE r8 r23
#define DECLARE r8 r24
#define DECLARE r8 r25
#define DECLARE r8 r26
#define DECLARE r8 r27
#define DECLARE r8 r28
#define DECLARE r8 r29
#define DECLARE r8 r30
#define DECLARE_r8_r31

/*
 *  r9 declare set
 */
#define DECLARE r9
#define DECLARE r9 r10
#define DECLARE r9 r11
#define DECLARE r9 r12
#define DECLARE r9 r13
#define DECLARE r9 r14
#define DECLARE r9 r15
#define DECLARE r9 r16
#define DECLARE r9 r17
#define DECLARE r9 r18
#define DECLARE r9 r19
#define DECLARE_r9_r20
```

salppc.inc 3/9/2001

```
#define DECLARE r9 r21
#define DECLARE r9 r22
#define DECLARE r9 r23
#define DECLARE r9 r24
#define DECLARE r9 r25
#define DECLARE r9 r26
#define DECLARE r9 r27
#define DECLARE r9 r28
#define DECLARE r9 r29
#define DECLARE r9 r30
#define DECLARE_r9_r31

/*
 *  r10 declare set
 */
#define DECLARE r10
#define DECLARE r10 r11
#define DECLARE r10 r12
#define DECLARE r10 r13
#define DECLARE r10 r14
#define DECLARE r10 r15
#define DECLARE r10 r16
#define DECLARE r10 r17
#define DECLARE r10 r18
#define DECLARE r10 r19
#define DECLARE r10 r20
#define DECLARE r10 r21
#define DECLARE r10 r22
#define DECLARE r10 r23
#define DECLARE r10 r24
#define DECLARE r10 r25
#define DECLARE r10 r26
#define DECLARE r10 r27
#define DECLARE r10 r28
#define DECLARE r10 r29
#define DECLARE r10 r30
#define DECLARE_r10_r31

/*
 *  r11 declare set
 */
#define DECLARE r11
#define DECLARE r11 r12
#define DECLARE r11 r13
#define DECLARE r11 r14
#define DECLARE r11 r15
#define DECLARE r11 r16
#define DECLARE r11 r17
#define DECLARE r11 r18
#define DECLARE r11 r19
#define DECLARE r11 r20
#define DECLARE r11 r21
#define DECLARE r11 r22
#define DECLARE r11 r23
#define DECLARE r11 r24
#define DECLARE r11 r25
#define DECLARE r11 r26
#define DECLARE r11 r27
#define DECLARE r11 r28
#define DECLARE r11 r29
#define DECLARE r11 r30
#define DECLARE_r11_r31

/*
 *  r12 declare set
 */
#define DECLARE_r12
```

```
#define DECLARE r12 r13
#define DECLARE r12 r14
#define DECLARE r12 r15
#define DECLARE r12 r16
#define DECLARE r12 r17
#define DECLARE r12 r18
#define DECLARE r12 r19
#define DECLARE r12 r20
#define DECLARE r12 r21
#define DECLARE r12 r22
#define DECLARE r12 r23
#define DECLARE r12 r24
#define DECLARE r12 r25
#define DECLARE r12 r26
#define DECLARE r12 r27
#define DECLARE r12 r28
#define DECLARE r12 r29
#define DECLARE r12 r30
#define DECLARE_r12_r31

/*
 *  r13 declare set
 */
#define DECLARE r13
#define DECLARE r13 r14
#define DECLARE r13 r15
#define DECLARE r13 r16
#define DECLARE r13 r17
#define DECLARE r13 r18
#define DECLARE r13 r19
#define DECLARE r13 r20
#define DECLARE r13 r21
#define DECLARE r13 r22
#define DECLARE r13 r23
#define DECLARE r13 r24
#define DECLARE r13 r25
#define DECLARE r13 r26
#define DECLARE r13 r27
#define DECLARE r13 r28
#define DECLARE r13 r29
#define DECLARE r13 r30
#define DECLARE_r13_r31

/*
 *  r14 declare set
 */
#define DECLARE r14
#define DECLARE r14 r15
#define DECLARE r14 r16
#define DECLARE r14 r17
#define DECLARE r14 r18
#define DECLARE r14 r19
#define DECLARE r14 r20
#define DECLARE r14 r21
#define DECLARE r14 r22
#define DECLARE r14 r23
#define DECLARE r14 r24
#define DECLARE r14 r25
#define DECLARE r14 r26
#define DECLARE r14 r27
#define DECLARE r14 r28
#define DECLARE r14 r29
#define DECLARE r14 r30
#define DECLARE_r14_r31

/*
 *  r15 declare set
```

```
 */
#define DECLARE r15
#define DECLARE r15 r16
#define DECLARE r15 r17
#define DECLARE r15 r18
#define DECLARE r15 r19
#define DECLARE r15 r20
#define DECLARE r15 r21
#define DECLARE r15 r22
#define DECLARE r15 r23
#define DECLARE r15 r24
#define DECLARE r15 r25
#define DECLARE r15 r26
#define DECLARE r15 r27
#define DECLARE r15 r28
#define DECLARE r15 r29
#define DECLARE r15 r30
#define DECLARE_r15_r31

/*
 *  r16 declare set
 */
#define DECLARE r16
#define DECLARE r16 r17
#define DECLARE r16 r18
#define DECLARE r16 r19
#define DECLARE r16 r20
#define DECLARE r16 r21
#define DECLARE r16 r22
#define DECLARE r16 r23
#define DECLARE r16 r24
#define DECLARE r16 r25
#define DECLARE r16 r26
#define DECLARE r16 r27
#define DECLARE r16 r28
#define DECLARE r16 r29
#define DECLARE r16 r30
#define DECLARE_r16_r31

/*
 *  r17 declare set
 */
#define DECLARE r17
#define DECLARE r17 r18
#define DECLARE r17 r19
#define DECLARE r17 r20
#define DECLARE r17 r21
#define DECLARE r17 r22
#define DECLARE r17 r23
#define DECLARE r17 r24
#define DECLARE r17 r25
#define DECLARE r17 r26
#define DECLARE r17 r27
#define DECLARE r17 r28
#define DECLARE r17 r29
#define DECLARE r17 r30
#define DECLARE_r17_r31

/*
 *  r18 declare set
 */
#define DECLARE r18
#define DECLARE r18 r19
#define DECLARE r18 r20
#define DECLARE r18 r21
#define DECLARE r18 r22
#define DECLARE_r18_r23
```

salppc.inc 3/9/2001

```
#define DECLARE_r18_r24
#define DECLARE_r18_r25
#define DECLARE_r18_r26
#define DECLARE_r18_r27
#define DECLARE_r18_r28
#define DECLARE_r18_r29
#define DECLARE_r18_r30
#define DECLARE_r18_r31

/*
 *  r19 declare set
 */
#define DECLARE_r19
#define DECLARE_r19_r20
#define DECLARE_r19_r21
#define DECLARE_r19_r22
#define DECLARE_r19_r23
#define DECLARE_r19_r24
#define DECLARE_r19_r25
#define DECLARE_r19_r26
#define DECLARE_r19_r27
#define DECLARE_r19_r28
#define DECLARE_r19_r29
#define DECLARE_r19_r30
#define DECLARE_r19_r31

/*
 *  FPR single precision declare set
 */
#define DECLARE_f0
#define DECLARE_f0_f1
#define DECLARE_f0_f2
#define DECLARE_f0_f3
#define DECLARE_f0_f4
#define DECLARE_f0_f5
#define DECLARE_f0_f6
#define DECLARE_f0_f7
#define DECLARE_f0_f8
#define DECLARE_f0_f9
#define DECLARE_f0_f10
#define DECLARE_f0_f11
#define DECLARE_f0_f12
#define DECLARE_f0_f13
#define DECLARE_f0_f14
#define DECLARE_f0_f15
#define DECLARE_f0_f16
#define DECLARE_f0_f17
#define DECLARE_f0_f18
#define DECLARE_f0_f19
#define DECLARE_f0_f20
#define DECLARE_f0_f21
#define DECLARE_f0_f22
#define DECLARE_f0_f23
#define DECLARE_f0_f24
#define DECLARE_f0_f25
#define DECLARE_f0_f26
#define DECLARE_f0_f27
#define DECLARE_f0_f28
#define DECLARE_f0_f29
#define DECLARE_f0_f30
#define DECLARE_f0_f31

/*
 *  FPR double precision declare set
 */
#define DECLARE_d0
#define DECLARE_d0_d1
```

```
#define DECLARE d0 d2
#define DECLARE d0 d3
#define DECLARE d0 d4
#define DECLARE d0 d5
#define DECLARE d0 d6
#define DECLARE d0 d7
#define DECLARE d0 d8
#define DECLARE d0 d9
#define DECLARE d0 d10
#define DECLARE d0 d11
#define DECLARE d0 d12
#define DECLARE d0 d13
#define DECLARE d0 d14
#define DECLARE d0 d15
#define DECLARE d0 d16
#define DECLARE d0 d17
#define DECLARE d0 d18
#define DECLARE d0 d19
#define DECLARE d0 d20
#define DECLARE d0 d21
#define DECLARE d0 d22
#define DECLARE d0 d23
#define DECLARE d0 d24
#define DECLARE d0 d25
#define DECLARE d0 d26
#define DECLARE d0 d27
#define DECLARE d0 d28
#define DECLARE d0 d29
#define DECLARE d0 d30
#define DECLARE_d0_d31

/*
 *  VMX declare set
 */
#define DECLARE v0
#define DECLARE v0 v1
#define DECLARE v0 v2
#define DECLARE v0 v3
#define DECLARE v0 v4
#define DECLARE v0 v5
#define DECLARE v0 v6
#define DECLARE v0 v7
#define DECLARE v0 v8
#define DECLARE v0 v9
#define DECLARE v0 v10
#define DECLARE v0 v11
#define DECLARE v0 v12
#define DECLARE v0 v13
#define DECLARE v0 v14
#define DECLARE v0 v15
#define DECLARE v0 v16
#define DECLARE v0 v17
#define DECLARE v0 v18
#define DECLARE v0 v19
#define DECLARE v0 v20
#define DECLARE v0 v21
#define DECLARE v0 v22
#define DECLARE v0 v23
#define DECLARE v0 v24
#define DECLARE v0 v25
#define DECLARE v0 v26
#define DECLARE v0 v27
#define DECLARE v0 v28
#define DECLARE v0 v29
#define DECLARE v0 v30
#define DECLARE_v0_v31
```

```
#endif                                  /* end SALPPC_INC */

/*
 *--------------------------------------------------------------
 *
 *                  END OF FILE salppc.inc
 *
 *--------------------------------------------------------------
 */
```

sve3_8bit.mac 2/23/2001

```
/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version   ---|
|--------------------------------------------------------------|
|                                                              |
|  File Name:    SVE3 8BIT.MAC                                 |
|  Description:  Sum the elements of 3 signed byte vectors     |
|                each of length N.                             |
|                                                              |
|  sve3_8bit ( char *A, char *B, char *C, long *SUM, int N )   |
|                                                              |
|  Restrictions:  A, B and C must all be 16-byte aligned.      |
|                 N must be a multiple of 16 and >= 16.        |
|                                                              |
|            Mercury Computer Systems, Inc.                    |
|            Copyright (c) 2000  All rights reserved           |
|                                                              |
|   Revision      Date     Engineer  Reason                    |
|   --------      ----     --------  --------                  |
|     0.0        000605      fpl    Created                    |
+-------------------------------------------------------------*/

#include "salppc.inc"

/**
 Input parameters
**/
#define A       r3
#define B       r4
#define C       r5
#define SUM     r6
#define N       r7

#define A0p     A
#define B0p     B
#define C0p     C
#define A1p     r8
#define B1p     r9
#define C1p     r10
#define index   r11

#define zero    v0
#define one     v1
#define a0      v2
#define a1      v3
#define b0      v4
#define b1      v5
#define c0      v6
#define c1      v7
#define sum0    v8
#define sum1    v9
#define sum2    v10

FUNC_PROLOG

ENTRY_5( sve3_8bit, A, B, C, SUM, N )

  USE_THRU_v10( VRSAVE_COND )

  LI( index, 0 )
        VXOR( zero, zero, zero )
  ADDIC C( N, N, -32 )
    LVX( a0, A0p, index )
      VSPLTISB( one, 1 )
    LVX( b0, B0p, index )
  ADDI( A1p, A0p, 16 )
        VXOR( sum0, sum0, sum0 )
```

```
  ADDI( B1p, B0p, 16 )
        VXOR( sum1, sum1, sum1 )
  ADDI( C1p, C0p, 16 )
        VXOR( sum2, sum2, sum2 )
  BLT( do16 )

LABEL( loop )
  ADDIC C( N, N, -32 )
    LVX( c0, C0p, index )
        VMSUMMBM( sum0, a0, one, sum0 )
    LVX( a1, A1p, index )
        VMSUMMBM( sum1, b0, one, sum1 )
    LVX( b1, B1p, index )
        VMSUMMBM( sum2, c0, one, sum2 )
    LVX( c1, C1p, index )
  ADDI( index, index, 32 )
        VMSUMMBM( sum0, a1, one, sum0 )
    LVX( a0, A0p, index )
        VMSUMMBM( sum1, b1, one, sum1 )
    LVX( b0, B0p, index )
        VMSUMMBM( sum2, c1, one, sum2 )
    BGE( loop )

  CMPWI( N, -32 )
    BEQ( combine )

LABEL( do16 )
    LVX( c0, C0p, index )
        VMSUMMBM( sum0, a0, one, sum0 )
        VMSUMMBM( sum1, b0, one, sum1 )
        VMSUMMBM( sum2, c0, one, sum2 )

LABEL( combine )
        VADDUWM( sum0, sum0, sum1 )
        VADDUWM( sum0, sum0, sum2 )
        VSUMSWS( sum0, sum0, zero )
      VSPLTW( sum0, sum0, 3 )
    STVEWX( sum0, 0, SUM )

  FREE THRU_v10( VRSAVE_COND )
  RETURN

FUNC_EPILOG
```

voter_sync.vhd 3/9/2001

```
-- ***********************************************************
--***********************************************************
--**
--**  Majority Voter/Sync Control logic TOP LEVEL Module: voter_sync.vhd
--**
--**  Description: This Module is the top level of the
--**  Majority Voter and Raceway Sync Logic
--**
--**  Author      : Steven Imperiali
--**  Date        : 7-05-2000
--**  Date        : 10-25-2000 Modified cable clock and sync
--**
--***********************************************************
--
-- This PLD handles the following functions:
--      1) Raceway clock source and skew control
--      2) Raceway sync generation
--      3) Majority voter logic
--      4) I2C reset logic
--      5) Inverter for the HS LED signal


LIBRARY IEEE;
USE IEEE.STD LOGIC 1164.ALL;
USE STD.TEXTIO.ALL;
use ieee.std logic arith.all;
use ieee.std_logic_unsigned.all;

ENTITY voter_sync IS
  PORT(
        clk 66 pal6       :IN     std logic;
        clk 33 pall       :IN     std logic;
        reset 0           :IN     std logic;
        x rst brd 0       :OUT    std logic;
        x rst brd 1       :OUT    std logic;
        pll rng sel       :OUT    std logic;
        pll freq sel      :OUT    std logic;
        fb sk sel         :OUT    std logic;
        fb dev by 2 0     :OUT    std logic;
        main sk sel0      :OUT    std logic;
        main sk sel1      :OUT    std logic;
        jk sk sel0        :OUT    std logic;
        jk sk sel1        :OUT    std logic;
        jx1 clk oe        :OUT    std logic;
        jx2 clk oe        :OUT    std logic;
        sw clk mode2_1    :IN     std logic vector(2 downto 1);
        mux clk sel0      :OUT    std logic;
        mux_clk_sel1      :OUT    std_logic;


        testn             :IN     std logic;
        tms0              :IN     std logic;
        rsync x nd0       :OUT    std logic;
        rsync x nd1       :OUT    std logic;
        rsync x nd2       :OUT    std logic;
        rsync x nd3       :OUT    std logic;
        rsync x pxb0      :OUT    std logic;
        rsync_x_xbar      :OUT    std_logic;

        nd0 resetreq 0    :IN     std logic;
        nd1 resetreq 0    :IN     std logic;
        nd2 resetreq 0    :IN     std logic;
        nd3 resetreq 0    :IN     std logic;
        pq resetreq_0     :IN     std logic;
        resetvote_0       :OUT    std_logic;

        nd0_ckstpreqnd0_0 :IN     std_logic;
```

voter_sync.vhd 3/9/2001

```
        nd0 ckstpreqnd1 0 :IN   std logic;
        nd0 ckstpreqnd2 0 :IN   std logic;
        nd0 ckstpreqnd3 0 :IN   std logic;
        nd0 ckstpreqpq 0  :IN   std logic;
        nd1 ckstpreqnd0 0 :IN   std logic;
        nd1 ckstpreqnd1 0 :IN   std logic;
        nd1 ckstpreqnd2 0 :IN   std logic;
        nd1 ckstpreqnd3 0 :IN   std logic;
        nd1 ckstpreqpq 0  :IN   std logic;
        nd2 ckstpreqnd0 0 :IN   std logic;
        nd2 ckstpreqnd1 0 :IN   std logic;
        nd2 ckstpreqnd2 0 :IN   std logic;
        nd2 ckstpreqnd3 0 :IN   std logic;
        nd2 ckstpreqpq 0  :IN   std logic;
        nd3 ckstpreqnd0 0 :IN   std logic;
        nd3 ckstpreqnd1 0 :IN   std logic;
        nd3 ckstpreqnd2 0 :IN   std logic;
        nd3 ckstpreqnd3 0 :IN   std logic;
        nd3 ckstpreqpq 0  :IN   std logic;
        pq ckstpreqnd0 0 :IN    std logic;
        pq ckstpreqnd1 0 :IN    std logic;
        pq ckstpreqnd2 0 :IN    std logic;
        pq ckstpreqnd3 0 :IN    std logic;
        pq ckstpreqpq_0  :IN    std logic;
        pq ckstopin 0    :OUT   std logic;
        nd0 ckstopin 0   :OUT   std logic;
        nd1 ckstopin 0   :OUT   std logic;
        nd2 ckstopin 0   :OUT   std logic;
        nd3_ckstopin_0   :OUT   std_logic;

        i2c_rst_0       :IN     std logic;
        sda             :INOUT  std logic;
        scl             :INOUT  std logic;
        pxb0 hs_led     :IN     std logic;
        hs_led          :OUT    std_logic
        );

END voter_sync;

ARCHITECTURE TOP_LEVEL_voter_sync OF voter_sync IS

--*****************************************************************
--*****************************************************************
--**  Component Declearation
--*****************************************************************
--*****************************************************************


COMPONENT m voter PORT(
        clk 66 pal6     :IN     std logic;
        reset 0         :IN     std logic;
        request0 0      :IN     std logic;
        request1 0      :IN     std logic;
        request2 0      :IN     std logic;
        request3 0      :IN     std logic;
        request4 0      :IN     std logic;
        healthy0 1      :IN     std logic;
        healthy1 1      :IN     std logic;
        healthy2 1      :IN     std logic;
        healthy3 1      :IN     std logic;
        healthy4 1      :IN     std logic;
        voteout_0       :OUT    std_logic);
END COMPONENT;


--*****************************************************************
```

voter_sync.vhd 3/9/2001

```
--** Signals to Connect All of the Components Together
--*****************************************************************

Signal healthy0 1       :std logic;
Signal healthy1 1       :std logic;
Signal healthy2 1       :std logic;
Signal healthy3 1       :std logic;
Signal healthy4_1       :std logic;
Signal sync d1          :std logic;
Signal sync d2          :std logic;
Signal sync d3          :std logic;
Signal nd0 ckstop_0, nd1_ckstop_0, nd2_ckstop_0, nd3_ckstop_0, pq_ckstop_0
:std logic;
Signal  g nd0 resetreq 0 :std logic;
Signal  g nd1 resetreq 0 :std logic;
Signal  g nd2 resetreq 0 :std logic;
Signal  g_nd3_resetreq_0 :std_logic;


BEGIN

--*****************************************************************
--**  Begin Architecture Here (Instantiations)
--*****************************************************************


nd0_ckstop voter : m_voter PORT Map(
        clk 66 pal6,
        reset 0,
        nd0 ckstpreqnd0 0,
        nd1 ckstpreqnd0 0,
        nd2 ckstpreqnd0 0,
        nd3 ckstpreqnd0 0,
        pq ckstpreqnd0_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        nd0_ckstop_0);


nd1_ckstop voter : m_voter PORT Map(
        clk 66 pal6,
        reset 0,
        nd0 ckstpreqnd1 0,
        nd1 ckstpreqnd1 0,
        nd2 ckstpreqnd1 0,
        nd3 ckstpreqnd1 0,
        pq ckstpreqnd1_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        nd1_ckstop_0);


nd2_ckstop voter : m_voter PORT Map(
        clk 66 pal6,
        reset 0,
        nd0 ckstpreqnd2 0,
        nd1 ckstpreqnd2 0,
        nd2_ckstpreqnd2_0,
```

```
        nd3 ckstpreqnd2 0,
        pq ckstpreqnd2_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        nd2_ckstop_0);


nd3_ckstop voter : m_voter PORT Map(
        clk 66 pal6,
        reset 0,
        nd0 ckstpreqnd3 0,
        nd1 ckstpreqnd3 0,
        nd2 ckstpreqnd3 0,
        nd3 ckstpreqnd3 0,
        pq ckstpreqnd3_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        nd3_ckstop_0);


pq_ckstop voter : m voter PORT Map(
        clk 66 pal6,
        reset 0,
        nd0 ckstpreqpq 0,
        nd1 ckstpreqpq 0,
        nd2 ckstpreqpq 0,
        nd3 ckstpreqpq 0,
        pq ckstpreqpq_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        pq_ckstop_0);


-- ####################################################
-- this section was added to force a board level reset when
-- the 8240 has a watchdog failure.

-- this should have been done by feeding the 8240's WDFAIL
-- to the reset PLD instead of forcing the 8240's resetreq
-- to drive all other resetrequests.

g nd0 resetreq 0 <= nd0 resetreq 0 AND pq resetreq 0;
g nd1 resetreq 0 <= nd1 resetreq 0 AND pq resetreq 0 ;
g nd2 resetreq 0 <= nd2 resetreq 0 AND pq resetreq 0 ;
g_nd3_resetreq_0 <= nd3_resetreq_0 AND pq_resetreq_0 ;

-- ####################################################


reset_req voter : m voter PORT Map(
        clk 66 pal6,
        reset 0,
        g nd0 resetreq 0,
        g nd1 resetreq 0,
        g nd2 resetreq 0,
        g_nd3_resetreq_0,
```

```
        pq resetreq_0,
        healthy0 1,
        healthy1 1,
        healthy2 1,
        healthy3 1,
        healthy4 1,
        resetvote_0);


        healthy0 1 <= nd0 ckstop 0;
        healthy1 1 <= nd1 ckstop 0;
        healthy2 1 <= nd2 ckstop 0;
        healthy3 1 <= nd3 ckstop 0;
        healthy4_1 <= pq_ckstop_0;

        nd0 ckstopin 0      <= nd0 ckstop 0;
        nd1 ckstopin 0      <= nd1 ckstop 0;
        nd2 ckstopin 0      <= nd2 ckstop 0;
        nd3 ckstopin 0      <= nd3 ckstop 0;
        pq_ckstopin_0       <= pq_ckstop_0;


WITH i2c_rst_0 SELECT

        sda <= clk_33_pal1 WHEN '0',
               'Z'         WHEN '1',
        'Z'     WHEN OTHERS;

WITH i2c_rst_0 SELECT

        scl <= clk_33_pal1 WHEN '0',
               'Z'         WHEN '1',
        'Z'     WHEN OTHERS;


        hs_led  <= NOT(pxb0_hs_led);


-- Sync Control
process(clk_66_pal6,reset_0)
BEGIN
        IF (reset 0 = '0') THEN
                sync d1         <= '1';
                sync d2         <= '1';
                sync d3         <= '1';
                rsync x nd0     <= '0';
                rsync x nd1     <= '0';
                rsync x nd2     <= '0';
                rsync x nd3     <= '0';
                rsync x pxb0    <= '0';
                rsync_x_xbar    <= '0';

        ELSIF (testn = '0' AND reset 0 = '1') THEN
                rsync x nd0     <= tms0;
                rsync x nd1     <= tms0;
                rsync x nd2     <= tms0;
                rsync x nd3     <= tms0;
                rsync x pxb0    <= '0';
                rsync_x_xbar    <= '0';

        ELSIF rising edge(clk 66 pal6) THEN
                sync d1 <= NOT(sync d1);
                sync_d2 <= (NOT(sync_d2) AND sync_d1 OR sync_d2 AND
```

```
                NOT(sync d1))
                sync d3 <= (NOT(NOT(sync d1) AND sync_d2));
                rsync x nd0     <= sync d3;
                rsync x nd1     <= sync d3;
                rsync x nd2     <= sync d3;
                rsync x nd3     <= sync d3;
                rsync x pxb0    <= sync d3;
                rsync_x_xbar    <= sync_d3;
        END IF;
        END process;


x rst brd 0 <= reset 0;
x_rst_brd_1 <= NOT(reset_0);


WITH sw clk mode2 1 SELECT
mux_clk_sel0     <=       '0'      WHEN     "00",     -- 66MHz local
                                    '0'     WHEN     "01",     -- 33MHz cable 1
                              '1' WHEN     "10",     -- 33MHz cable 2
                                    '0'     WHEN     "11",     -- 66 MHz local
                                    '1'     WHEN OTHERS;


WITH sw clk mode2 1 SELECT
mux_clk_sel1     <=       '0'      WHEN     "00",
                                    '1'     WHEN     "01",
                              '1' WHEN     "10",
                                    '0'     WHEN     "11",
                              '1'      WHEN OTHERS;

WITH sw clk mode2 1 SELECT
fb_dev_by_2_0    <=       '0'      WHEN     "00",
                                    'Z'     WHEN     "01",
                              'Z' WHEN     "10",
                                    '0'     WHEN     "11",
                              '1'      WHEN OTHERS;

WITH sw clk_mode2 1 SELECT
jx1_clk_oe       <=       '1'      WHEN     "00",
                              '1' WHEN     "01",
                          '1'      WHEN     "10",
                              '1' WHEN     "11",
                          '1'      WHEN OTHERS;

WITH sw clk_mode2 1 SELECT
jx2_clk_oe       <=       '1'      WHEN     "00",
                              '1' WHEN     "01",
                          '1'      WHEN     "10",
                              '1' WHEN     "11",
                          '1'      WHEN OTHERS;


WITH sw clk_mode2 1 SELECT
pll_rng_sel      <=       '1'      WHEN     "00",
                                    '1'     WHEN     "01",
                              '1' WHEN     "10",
                                    '1'     WHEN     "11",
                              '1'      WHEN OTHERS;

WITH sw clk mode2 1 SELECT
pll_freq_sel     <=       'Z'      WHEN     "00",
                                    '0'     WHEN     "01",
                              '0' WHEN     "10",
                                    'Z'     WHEN     "11",
```

```
                            '1'     WHEN OTHERS;

-- select 0 skew for all modes

WITH sw_clk mode2_1 SELECT
        fb_sk_sel  <=     'Z'     WHEN    "00",
                                  'Z'     WHEN    "01",
                          'Z'     WHEN    "10",
                                  'Z'     WHEN    "11",
                          '1'     WHEN OTHERS;

WITH sw_clk mode2 1 SELECT
        main_sk_sel0 <= 'Z'       WHEN    "00",
                                  'Z'     WHEN    "01",
                          'Z'     WHEN    "10",
                                  'Z'     WHEN    "11",
                          '1'     WHEN OTHERS;

WITH sw_clk mode2 1 SELECT
        main_sk_sel1 <= 'Z'       WHEN    "00",
                                  'Z'     WHEN    "01",
                          'Z'     WHEN    "10",
                                  'Z'     WHEN    "11",
                          '1'     WHEN OTHERS;

WITH sw_clk mode2 1 SELECT
        jk_sk_sel0  <=    'Z'     WHEN    "00",
                                  'Z'     WHEN    "01",
                          'Z'     WHEN    "10",
                                  'Z'     WHEN    "11",
                          '1'     WHEN OTHERS;

WITH sw_clk mode2 1 SELECT
        jk_sk_sel1  <=    'Z'     WHEN    "00",
                                  'Z'     WHEN    "01",
                          'Z'     WHEN    "10",
                                  'Z'     WHEN    "11",
                          '1'     WHEN OTHERS;

END TOP_LEVEL_voter_sync;
```

zdotpr4_vmx.k 2/23/2001

```
/*-----------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version ---|
|-------------------------------------------------------------|
|                                                             |
|  File Name:     ZDOTPR4 VMX.K                               |
|  Description:   CPP Source code for Vector Single Precision |
|                 Split Complex Dot Product given that input  |
|                 vectors are relivatively unaligned.         |
|                                                             |
|  Entry/params:  ZDOTPR4 VMX (A, I, B, J, C, N)              |
|                 _ZIDOTPR4_VMX (A, I, B, J, C, N)            |
|                                                             |
|  Formula: C[0] = sum (A->realp[mI]*B->realp[mJ]             |
|                    -/+ A->imagp[mI]*B->imagp[mJ])           |
|           C[1] = sum (A->realp[mI]*B->imagp[mJ]             |
|                    +/- A->imagp[mI]*B->realp[mJ])           |
|                  for m=0 to N-1                             |
|                                                             |
|              Mercury Computer Systems, Inc.                 |
|              Copyright (c) 2000  All rights reserved        |
|                                                             |
|   Revision    Date      Engineer Reason                     |
|   --------    ----      -------- --------                   |
|     0.0       000608    fpl      Created (from zdotpr vmx.k)|
+------------------------------------------------------------*/
#include "salppc.inc"
/**
 ESAL CPP definitions
**/
#undef  FUNC ENTRY
#undef  LOAD A
#undef  LOAD B
#undef  SUFFIX

#if defined( VMX_SAL )

#define  FUNC ENTRY          zdotpr4 vmx
#define  FUNC CONJ ENTRY    _zidotpr4 vmx
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label

#elif defined( VMX_NN )

#define  FUNC ENTRY          zdotpr4 vmx nn
#define  FUNC CONJ ENTRY    _zidotpr4_vmx_nn
#define  LOAD A( vT, rA, rB )  LVXL( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVXL( vT, rA, rB )
#define  SUFFIX( label )  label##_nn

#elif defined( VMX_NC )

#define  FUNC ENTRY          zdotpr4 vmx nc
#define  FUNC CONJ ENTRY    _zidotpr4_vmx_nc
#define  LOAD A( vT, rA, rB )  LVXL( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_nc

#elif defined( VMX_CN )

#define  FUNC ENTRY          zdotpr4 vmx cn
#define  FUNC CONJ ENTRY    _zidotpr4 vmx cn
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVXL( vT, rA, rB )
#define  SUFFIX( label )  label##_cn

#elif defined( VMX_CC )
```

zdotpr4_vmx.k 2/23/2001

```
#define  FUNC ENTRY          zdotpr4 vmx cc
#define  FUNC CONJ ENTRY    _zidotpr4 vmx cc
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_cc

#else
#error  YOU MUST DEFINE VMX_xxx, where x = C or N

#endif

#define VREGSAVE_COND  VRSAVE_COND  /* defined as 7 in salppc.inc */

/**
 Local CPP definitions
**/
#define NMASK2 0x8
#define NMASK1 0x4
#define NSHIFT 4
#define ADDRESS_INCREMENT 16

/**
 Input args
**/
#define A   r3
#define I   r4
#define B   r5
#define J   r6
#define C   r7
#define N   r8
#define EFLAG r9

/**
 Split complex parameters
**/
#define Ar0 A
#define Ai0 r10
#define Br0 B
#define Bi0 r11
#define Cr  C
#define Ci  r12

/**
 Local registers
**/
#define count r4
#define rtmp0 r4
#define rtmp1 r13

#define Ar1 r13
#define Ai1 r14
#define Ar2 r15
#define Ai2 r16
#define Ar3 r17
#define Ai3 r18

#define Br1 r19
#define Bi1 r20
#define Br2 r21
#define Bi2 r22
#define Br3 r23
#define Bi3 r24
#define aoffset r25
#define coffset r25
#define boffset r26
#define addr_incr r27
```

zdotpr4_vmx.k 2/23/2001

```
/**
 VMX registers
**/
#define rsumr v0
#define rsumi v1
#define isumr v2
#define isumi v3
#define rsum0 v4
#define rsum1 v5
#define isum0 v6
#define isum1 v7

#define ar0 v4
#define ai0 v5
#define ar1 v6
#define ai1 v7
#define ar2 v8
#define ai2 v9
#define ar3 v10
#define ai3 v11

#define br0 v12
#define bi0 v13
#define br1 v14
#define bi1 v15
#define br2 v16
#define bi2 v17
#define br3 v18
#define bi3 v19
#define apC v20

#define atr0 v21
#define ati0 v22
#define atr1 v23
#define ati1 v24
#define atr2 v25
#define ati2 v26
#define atr3 v27
#define ati3 v28

/**
 FPU registers
**/
#define far     f0
#define fbr     f1
#define fai     f2
#define fbi     f3
#define frsumr  f4
#define frsumi  f5
#define fisumi  f6
#define fisumr  f7
#define frsum   f8
#define fisum   f9
#define rsum_vmx f10
#define isum_vmx f11

/**
 Begin code text, Save some registers
 Here for conjugate inner product
**/
U_ENTRY( FUNC CONJ_ENTRY )
   MR(rtmp0, Cr)
   MR(Cr, Ci)
   MR(Ci, rtmp0)
   MR(rtmp0, Br0)
   MR(Br0, Bi0)
   MR(Bi0, rtmp0)
```

zdotpr4_vmx.k 2/23/2001

```
/**
 Here for normal inner product
**/
FUNC PROLOG
U_ENTRY( FUNC ENTRY )
   DECLARE f0 f11
   DECLARE r3 r27
   DECLARE_v0_v28
/**
 Initial setup code
**/
   SAVE r13 r27
   USE THRU v28( VREGSAVE_COND )
   LFS( frsumr, Ar0, 0 )
   FSUBS(frsumr, frsumr, frsumr)
   FMR(frsumi, frsumr)
   FMR(fisumr, frsumr)
   FMR(fisumi, frsumr)
   FMR(rsum vmx, frsumr)
   FMR(isum_vmx, frsumr)
/**
 Process unaligned vector section first
**/
LABEL( SUFFIX( cont ) )
   GET_VMX UNALIGNED_COUNT( count, Br0 )
   LI( aoffset, 0 )
   LI( boffset, 0 )
   BEQ( SUFFIX( aligned ) )
   SUB( N, N, count )        /* adjust N for after loop */
/**
 Here to do first 1 to 3 points using standard FP
 Store result for later post_loop processing
**/
  LFSX( far, Ar0, aoffset )
  LFSX( fai, Ai0, aoffset )
  DECR C( count )
  LFSX( fbr, Br0, boffset )
  LFSX( fbi, Bi0, boffset )
  FMULS( frsumr, far, fbr )
  FMULS( frsumi, fai, fbi )
  FMULS( fisumi, far, fbi )
  FMULS( fisumr, fai, fbr )
  ADDI( Ar0, Ar0, 4 )
  ADDI( Ai0, Ai0, 4 )
  ADDI( Br0, Br0, 4 )
  ADDI( Bi0, Bi0, 4 )
  BEQ( SUFFIX( aligned ) )
/**
 Loop does 1 or 2 more sum updates
**/
LABEL( SUFFIX( pre_loop ) )
   LFSX( far, Ar0, aoffset )
   LFSX( fai, Ai0, aoffset )
   DECR C( count )
   LFSX( fbr, Br0, boffset )
   LFSX( fbi, Bi0, boffset )
   FMADDS( frsumr, far, fbr, frsumr )
   ADDI( Ar0, Ar0, 4 )
   FMADDS( frsumi, fai, fbi, frsumi )
   ADDI( Ai0, Ai0, 4 )
   FMADDS( fisumi, far, fbi, fisumi )
   ADDI( Br0, Br0, 4 )
   FMADDS( fisumr, fai, fbr, fisumr )
   ADDI( Bi0, Bi0, 4 )
   BNE( SUFFIX( pre_loop) )
/**
 Here for VMX aligned loop code
```

zdotpr4_vmx.k 2/23/2001

```
 Prepare for loop entry: assign loop pointers, counters
**/
LABEL( SUFFIX( aligned ) )
   SRWI C( count, N, 4 )  /* 16 per trip */
   LVSL( apC, Ar0, aoffset )
   LI( aoffset, 0 )
   LI( boffset, 0 )

   ADDI( Ar1, Ar0, 16 )
   VXOR( rsumr, rsumr, rsumr )
   ADDI( Ar2, Ar0, 32 )
   ADDI( Ar3, Ar0, 48 )

   ADDI( Ai1, Ai0, 16 )
   VXOR( isumi, isumi, isumi )
   ADDI( Ai2, Ai0, 32 )
   ADDI( Ai3, Ai0, 48 )

   ADDI( Br1, Br0, 16 )
   VXOR( rsumi, rsumi, rsumi )
   ADDI( Br2, Br0, 32 )
   ADDI( Br3, Br0, 48 )

   ADDI( Bi1, Bi0, 16 )
   ADDI( Bi2, Bi0, 32 )
   VXOR( isumr, isumr, isumr )
   ADDI( Bi3, Bi0, 48 )
   BEQ( SUFFIX(two_left) )
/**
 Loop windin section
**/
   LOAD A( atr0, Ar0, aoffset )
   LOAD A( ati0, Ai0, aoffset )
   LOAD A( atr1, Ar1, aoffset )
   LOAD_A( ati1, Ai1, aoffset )

   LOAD A( atr2, Ar2, aoffset )
   LOAD A( ati2, Ai2, aoffset )
   VPERM( ar0, atr0, atr1, apC )
   LOAD B( br0, Br0, boffset )
   LOAD B( bi0, Bi0, boffset )
   DECR C( count )
   VPERM( ai0, ati0, ati1, apC )
   LOAD B( br1, Br1, boffset )
   VPERM( ar1, atr1, atr2, apC )
   LOAD A( atr3, Ar3, aoffset )
   BR( SUFFIX( mid_loop ) )
/**
 Top of vector loop
**/
LABEL( SUFFIX( loop ) )
/* { */
    LOAD A( atr2, Ar2, aoffset )
    VMADDFP( rsumr, ar3, br3, rsumr )
    LOAD A( ati2, Ai2, aoffset )
    VPERM( ar0, atr0, atr1, apC ) /* uses last pass value */
    VMADDFP( rsumi, ai3, bi3, rsumi )
    LOAD B( br0, Br0, boffset )
    LOAD B( bi0, Bi0, boffset )
    DECR C( count )
    VPERM( ai0, ati0, ati1, apC )
    LOAD B( br1, Br1, boffset )
    VPERM( ar1, atr1, atr2, apC )
    VMADDFP( isumi, ar3, bi3, isumi )
    LOAD A( atr3, Ar3, aoffset )
    VMADDFP( isumr, ai3, br3, isumr )
/**
```

```
 Loop entry
**/
LABEL( SUFFIX( mid loop ) )
    VMADDFP( rsumr, ar0, br0, rsumr )
    VPERM( ai1, ati1, ati2, apC )
    VMADDFP( rsumi, ai0, bi0, rsumi )
    LOAD A( ati3, Ai3, aoffset )
    VMADDFP( isumr, ai0, br0, isumr )
    LOAD B( bi1, Bi1, boffset )
    ADDI( aoffset, aoffset, 64 )
    VPERM( ar2, atr2, atr3, apC )
    VMADDFP( isumi, ar0, bi0, isumi )
    LOAD B( br2, Br2, boffset )
    VMADDFP( rsumr, ar1, br1, rsumr )
    LOAD B( bi2, Bi2, boffset )
    VMADDFP( isumr, ai1, br1, isumr )
/**
 Loop exit
**/
    VPERM( ai2, ati2, ati3, apC )
    BEQ( SUFFIX(loop exit ) )
    LOAD A( atr0, Ar0, aoffset )
    VMADDFP( rsumi, ai1, bi1, rsumi )
    LOAD A( ati0, Ai0, aoffset )
    VMADDFP( isumi, ar1, bi1, isumi )
    LOAD B( br3, Br3, boffset )
    VPERM( ar3, atr3, atr0, apC )
    VMADDFP( rsumr, ar2, br2, rsumr )
    LOAD A( atr1, Ar1, aoffset )
    VMADDFP( rsumi, ai2, bi2, rsumi )
    VPERM( ai3, ati3, ati0, apC )
    VMADDFP( isumi, ar2, bi2, isumi )
    LOAD B( bi3, Bi3, boffset )
    ADDI( boffset, boffset, 64 )
    LOAD A( ati1, Ai1, aoffset )
    VMADDFP( isumr, ai2, br2, isumr )
/* } */
    BR( SUFFIX( loop ) )
/**
 windout section
**/
LABEL( SUFFIX(loop exit ) )
   LOAD A( atr0, Ar0, aoffset )
   VMADDFP( rsumi, ai1, bi1, rsumi )
   LOAD A( ati0, Ai0, aoffset )
   VMADDFP( isumi, ar1, bi1, isumi )
   LOAD B( br3, Br3, boffset )
   VPERM( ar3, atr3, atr0, apC )
   VMADDFP( rsumr, ar2, br2, rsumr )
   VMADDFP( rsumi, ai2, bi2, rsumi )
   VPERM( ai3, ati3, ati0, apC )
   VMADDFP( isumi, ar2, bi2, isumi )
   LOAD B( bi3, Bi3, boffset )
   ADDI( boffset, boffset, 64 )
   VMADDFP( isumr, ai2, br2, isumr )
   VMADDFP( rsumr, ar3, br3, rsumr )
   VMADDFP( rsumi, ai3, bi3, rsumi )
   VMADDFP( isumi, ar3, bi3, isumi )
   VMADDFP( isumr, ai3, br3, isumr )
/**
 Remaining sum updates
**/
LABEL( SUFFIX(two_left) )
   ANDI_C( count, N, 0x8 )  /* bit 3 */
   BEQ( SUFFIX(one_left ) )

   LOAD_B( br0, Br0, boffset )
```

```
   LOAD B( bi0, Bi0, boffset )
   LOAD B( br1, Br1, boffset )
   LOAD B( bi1, Bi1, boffset )
   ADDI( boffset, boffset, 32 )

   LOAD A( atr0, Ar0, aoffset )
   LOAD A( ati0, Ai0, aoffset )
   LOAD A( atr1, Ar1, aoffset )
   LOAD A( ati1, Ai1, aoffset )
   LOAD A( atr2, Ar2, aoffset )
   LOAD A( ati2, Ai2, aoffset )
   ADDI( aoffset, aoffset, 32 )

   VPERM( ar0, atr0, atr1, apC ) /* uses last pass value */
   VPERM( ai0, ati0, ati1, apC )
   VPERM( ar1, atr1, atr2, apC )
   VPERM( ai1, ati1, ati2, apC )

   VMADDFP( rsumr, ar0, br0, rsumr )
   VMADDFP( rsumi, ai0, bi0, rsumi )
   VMADDFP( isumr, ai0, br0, isumr )
   VMADDFP( isumi, ar0, bi0, isumi )

   VMADDFP( rsumr, ar1, br1, rsumr )
   VMADDFP( isumr, ai1, br1, isumr )
   VMADDFP( rsumi, ai1, bi1, rsumi )
   VMADDFP( isumi, ar1, bi1, isumi )
   VMR(atr3, atr1)
   VMR(ati3, ati1)

LABEL( SUFFIX(one_left) )
   ANDI_C( count, N, 0x4 )   /* bit 2 */
   BEQ( SUFFIX(combine ) )

   LOAD B( br0, Br0, boffset )
   LOAD B( bi0, Bi0, boffset )
   ADDI( boffset, boffset, 16 )

   LOAD A( atr0, Ar0, aoffset )
   LOAD A( ati0, Ai0, aoffset )
   LOAD A( atr1, Ar1, aoffset )
   LOAD A( ati1, Ai1, aoffset )
   ADDI( aoffset, aoffset, 16 )

   VPERM( ar0, atr0, atr1, apC ) /* uses last pass value */
   VPERM( ai0, ati0, ati1, apC )

   VMADDFP( rsumr, ar0, br0, rsumr )
   VMADDFP( rsumi, ai0, bi0, rsumi )
   VMADDFP( isumr, ai0, br0, isumr )
   VMADDFP( isumi, ar0, bi0, isumi )
/**
 combine partial sums, permute, write out results
**/
LABEL( SUFFIX(combine) )
  VSUBFP( rsumr, rsumr, rsumi ) /* rsumr = rsumr - rsumi */
  VADDFP( isumi, isumi, isumr )
/**
 8 bytes/cycle shuffle:
 real/imag logic should be intermixed for efficiency
**/
  VMRGHW(rsum0, rsumr, rsumr)
  ANDI C( addr incr, N, 0x3 )
  VMRGHW(isum0, isumi, isumi)
  VMRGLW(rsum1, rsumr, rsumr)
  SUB( addr incr, N, addr incr ) /* offset index for remainders */
  VMRGLW(isum1, isumi, isumi)
```

```
  VADDFP( rsum0, rsum1, rsum0 )
  SLWI(addr incr, addr incr, 2) /* byte offset */
  VADDFP( isum0, isum1, isum0 )

  VMRGHW(rsum1, rsum0, rsum0)
  ADD(Ar0, Ar0, addr incr)
  VMRGHW(isum1, isum0, isum0)
  ADD(Ai0, Ai0, addr incr)
  VMRGLW(rsum0, rsum0, rsum0)
  ADD(Br0, Br0, addr incr)
  VMRGLW(isum0, isum0, isum0)
  ADD(Bi0, Bi0, addr incr)
  VADDFP( rsumr, rsum1, rsum0 )
  LI(coffset, 0) /* needed for output */
  VADDFP( isumi, isum1, isum0 )
/**
 4 byte stores
**/
  STVEWX( rsumr, Cr, coffset )
  STVEWX( isumi, Ci, coffset )
/**
 Remainders of 1-3 more to do
**/
  ANDI_C( N, N, 3 )
  LFS( rsum vmx, Cr, 0 )
  LFS( isum vmx, Ci, 0 )
  BEQ( SUFFIX( scaler_vmx_combine ) )
/**
 Here to do last 1-3 points using standard FP
**/
LABEL( SUFFIX( post_loop ) )
   LFS( far, Ar0, 0 )
   LFS( fai, Ai0, 0 )
   DECR_C( N )
   LFS( fbr, Br0, 0 )
   LFS( fbi, Bi0, 0 )
   FMADDS( frsumr, far, fbr, frsumr )
   FMADDS( frsumi, fai, fbi, frsumi )
   FMADDS( fisumi, far, fbi, fisumi )
   FMADDS( fisumr, fai, fbr, fisumr )
   ADDI(Ar0, Ar0, 4)
   ADDI(Br0, Br0, 4)
   ADDI(Ai0, Ai0, 4)
   ADDI(Bi0, Bi0, 4)
   BNE( SUFFIX( post_loop) )
/**
 Write out result
**/
LABEL( SUFFIX( scaler vmx combine ) )
  FSUBS( frsum, frsumr, frsumi ) /* rsumr = rsumr - rsumi */
  FADDS( fisum, fisumi, fisumr )
  FADDS( frsum, frsum, rsum vmx )
  FADDS( fisum, fisum, isum_vmx )
  STFS( frsum, Cr, 0 )
  STFS( fisum, Ci, 0 )
/**
 return
**/
LABEL( SUFFIX(ret) )
   FREE THRU v28( VREGSAVE_COND )
   REST r13_r27
   RETURN
FUNC_EPILOG
```

```
zdotpr4_vmx.mac                                                  2/23/2001

/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version   ---|
|--------------------------------------------------------------|
|                                                              |
|  File Name:     ZDOTPR4 VMX.MAC                              |
|  Description:   Vector Single Precision Complex Dot Product  |
|                 CPP dummy file for unaligned vector processing|
|                                                              |
|  Entry/params:  ZDOTPR4 VMX (A, I, B, J, C, N)               |
|  Formula: C[0] = sum (A->realp[mI]*B->realp[mJ]              |
|                      - A->imagp[mI]*B->imagp[mJ])            |
|           C[1] = sum (A->realp[mI]*B->imagp[mJ]              |
|                      + A->imagp[mI]*B->realp[mJ])            |
|                      for m=0 to N-1                          |
|                                                              |
|              Mercury Computer Systems, Inc.                  |
|              Copyright (c) 1998  All rights reserved         |
|                                                              |
|   Revision  Date   Engineer  Reason                          |
|   --------  ----   --------  --------                        |
|     0.0    000607    fpl    Created (from zdotpr vmx.mac)    |
+--------------------------------------------------------------*/
#if defined( BUILD_MAX )

#undef VMX SAL
#undef VMX NN
#undef VMX NC
#undef VMX CN
#undef VMX_CC

#if !defined( COMPILE_ESAL_JUMP TABLE )
                              /* 1 variant: _zdotpr4_vmx() */
#define VMX SAL
#include "zdotpr4_vmx.k"

#else                         /* 5 variants based on ESAL flag */
#define VMX NN
#include "zdotpr4_vmx.k"

#undef VMX NN
#define VMX NC
#include "zdotpr4_vmx.k"

#undef VMX NC
#define VMX CN
#include "zdotpr4_vmx.k"

#undef VMX CN
#define VMX CC
#include "zdotpr4_vmx.k"
#undef VMX_CC
#endif                      /* end COMPILE_ESAL_JUMP_TABLE */

#endif                      /* end BUILD_MAX */
```

zdotpr_vmx.k 2/23/2001

```
/*-------------------------------------------------------------+
|---  MC Standard Algorithms -- PPC Macro language Version  ---|
|--------------------------------------------------------------|
|                                                              |
|  File Name:    ZDOTPR.K                                      |
|  Description:  CPP Source code for Vector Single Precision   |
|                Split Complex Dot Product                     |
|  Entry/params: ZDOTPR (A, I, B, J, C, N)                     |
|                ZIDOTPR (A, I, B, J, C, N)                    |
|                                                              |
|  Formula: C[0] = sum (A->realp[mI]*B->realp[mJ]              |
|                       -/+ A->imagp[mI]*B->imagp[mJ])         |
|           C[1] = sum (A->realp[mI]*B->imagp[mJ]              |
|                       +/- A->imagp[mI]*B->realp[mJ])         |
|                      for m=0 to N-1                          |
|                                                              |
|               Mercury Computer Systems, Inc.                 |
|               Copyright (c) 1998  All rights reserved        |
|                                                              |
|   Revision   Date     Engineer Reason                        |
|   --------   ----     -------- --------                      |
|     0.0      981215   fpl      Created                       |
|     0.1      990310   fpl      Integrated with 750 library   |
|     0.2      000131   jfk      salppc.inc changes            |
|     0.3      000223   fpl      Fixed pre-loop bug            |
|     0.4      000717   fpl      Added dsts, removed LVXLs     |
+-------------------------------------------------------------*/
#include "salppc.inc"

/**
 ESAL CPP definitions
**/
#undef  FUNC CONJ ENTRY
#undef  FUNC ENTRY
#undef  LOAD A
#undef  LOAD B
#undef  SUFFIX

#if defined( VMX_SAL )

#define  FUNC ENTRY          zdotpr vmx
#define  FUNC CONJ ENTRY    _zidotpr_vmx
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label
#undef   DSTA( ptr, control )
#undef   DSTB( ptr, control )
#define  DSTA( ptr, control )
#define  DSTB( ptr, control )
#undef   DST_ENABLE

#elif defined( VMX_NN )

#define  FUNC ENTRY          zdotpr vmx nn
#define  FUNC CONJ ENTRY    _zidotpr_vmx_nn
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_nn
#undef   DSTA( ptr, control )
#undef   DSTB( ptr, control )
#define  DSTA( ptr, control )
#define  DSTB( ptr, control )
#undef   DST_ENABLE

#elif defined( VMX_NC )

#define  FUNC_ENTRY         _zdotpr_vmx_nc
```

zdotpr_vmx.k 2/23/2001

```
#define  FUNC CONJ ENTRY     _zidotpr_vmx_nc
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_nc
#undef   DSTA( ptr, control )
#undef   DSTB( ptr, control )
#define  DSTA( ptr, control ) DST( ptr, control, 0 ) \
                              ADDI( ptr, ptr, 64 )
#define  DSTB( ptr, control )
#define DST_ENABLE

#elif defined( VMX_CN )

#define  FUNC ENTRY          zdotpr vmx cn
#define  FUNC CONJ ENTRY     _zidotpr_vmx_cn
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_cn
#undef   DSTA( ptr, control )
#undef   DSTB( ptr, control )
#define  DSTA( ptr, control )
#define  DSTB( ptr, control ) DST( ptr, control, 0 ) \
                               ADDI( ptr, ptr, 64 )
#define DST_ENABLE

#elif defined( VMX_CC )

//#define  FUNC ENTRY          zdotpr vmx_cc
#define  FUNC ENTRY          zdotpr vmx
#define  FUNC CONJ ENTRY     _zidotpr_vmx_cc
#define  LOAD A( vT, rA, rB )  LVX( vT, rA, rB )
#define  LOAD B( vT, rA, rB )  LVX( vT, rA, rB )
#define  SUFFIX( label )  label##_cc
#undef   DSTA( ptr, control )
#undef   DSTB( ptr, control )
#define  DSTA( ptr, control )
#define  DSTB( ptr, control )
#undef   DST_ENABLE

#else
#error  YOU MUST DEFINE VMX_xxx, where x = C or N

#endif

#define VREGSAVE_COND  VRSAVE_COND  /* defined as 7 in salppc.inc */

/**
 Local CPP definitions
**/
#define NMASK2 0x8
#define NMASK1 0x4
#define NSHIFT 4
#define ADDRESS_INCREMENT 16

/**
 Input args
**/
#define A   r3
#define I   r4
#define B   r5
#define J   r6
#define C   r7
#define N   r8
#define EFLAG r9

/**
 Split complex parameters
```

```
zdotpr_vmx.k                                                    2/23/2001

**/
#define Ar0 A
#define Ai0 r10
#define Br0 B
#define Bi0 r11
#define Cr  C
#define Ci  r12

/**
 Local registers
**/
#define count r4
#define rtmp0 r4
#define rtmp1 r13

#define dst stride r13
#define num_blocks r14
#define Ar1 r13
#define Ai1 r14
#define Ar2 r15
#define Ai2 r16
#define Ar3 r17
#define Ai3 r18

#define Br1 r19
#define Bi1 r20
#define Br2 r21
#define Bi2 r22
#define Br3 r23
#define Bi3 r24
#define ptr offset0 r25
#define ptr offset1 r26
#define addr incr r27
#define dst rptr r28
#define dst iptr r29
#define dst_control r30

/**
 VMX registers
**/
#define rsumr v0
#define rsumi v1
#define isumr v2
#define isumi v3
#define rsum0 v4
#define rsum1 v5
#define isum0 v6
#define isum1 v7

#define ar0 v4
#define ai0 v5
#define ar1 v6
#define ai1 v7
#define ar2 v8
#define ai2 v9
#define ar3 v10
#define ai3 v11

#define br0 v12
#define bi0 v13
#define br1 v14
#define bi1 v15
#define br2 v16
#define bi2 v17
#define br3 v18
#define bi3 v19
/**
```

```
 FPU registers
**/
#define far     f0
#define fbr     f1
#define fai     f2
#define fbi     f3
#define frsumr  f4
#define frsumi  f5
#define fisumi  f6
#define fisumr  f7
#define frsum   f8
#define fisum   f9
#define rsum vmx f10
#define isum_vmx f11

/**
 Begin code text, Save some registers
 Here for conjugate inner product
**/
U_ENTRY( FUNC CONJ_ENTRY )
   MR(rtmp0, Cr)
   MR(Cr, Ci)
   MR(Ci, rtmp0)
   MR(rtmp0, Br0)
   MR(Br0, Bi0)
   MR(Bi0, rtmp0)
/**
 Here for normal inner product
**/
U_ENTRY( FUNC ENTRY )
   DECLARE f0 f11
   DECLARE r3 r30
   DECLARE_v0_v19
/**
 Initial setup code
**/
   SAVE r13 r30
   USE THRU v19( VREGSAVE_COND )
   LFS( frsumr, Ar0, 0 )
   FSUBS(frsumr, frsumr, frsumr)
   FMR(frsumi, frsumr)
   FMR(fisumr, frsumr)
   FMR(fisumi, frsumr)
   FMR(rsum vmx, frsumr)
   FMR(isum_vmx, frsumr)
/**
 Process unaligned vector section first
**/
LABEL( SUFFIX( cont ) )
   GET_VMX UNALIGNED COUNT( count, Ar0 )
   LI( ptr offset0, 0 )
   BEQ( SUFFIX( aligned ) )
   SUB( N, N, count )       /* adjust N for after loop */
/**
 Here to do first 1 to 3 points using standard FP
 Store result for later post_loop processing
**/
  LFSX( far, Ar0, ptr offset0 )
  LFSX( fai, Ai0, ptr_offset0 )
  DECR C( count )
  LFSX( fbr, Br0, ptr offset0 )
  LFSX( fbi, Bi0, ptr_offset0 )
  FMULS( frsumr, far, fbr )
  FMULS( frsumi, fai, fbi )
  FMULS( fisumi, far, fbi )
  FMULS( fisumr, fai, fbr )
  ADDI( Ar0, Ar0, 4 )
```

```
  ADDI( Ai0, Ai0, 4 )
  ADDI( Br0, Br0, 4 )
  ADDI( Bi0, Bi0, 4 )
  BEQ( SUFFIX( aligned ) )
/**
 Loop does 1 or 2 more sum updates
**/
LABEL( SUFFIX( pre_loop ) )
   LFSX( far, Ar0, ptr offset0 )
   LFSX( fai, Ai0, ptr_offset0 )
   DECR C( count )
   LFSX( fbr, Br0, ptr offset0 )
   LFSX( fbi, Bi0, ptr offset0 )
   FMADDS( frsumr, far, fbr, frsumr )
   ADDI( Ar0, Ar0, 4 )
   FMADDS( frsumi, fai, fbi, frsumi )
   ADDI( Ai0, Ai0, 4 )
   FMADDS( fisumi, far, fbi, fisumi )
   ADDI( Br0, Br0, 4 )
   FMADDS( fisumr, fai, fbr, fisumr )
   ADDI( Bi0, Bi0, 4 )
   BNE( SUFFIX( pre_loop) )
/**
 Here for VMX aligned loop code
 Prepare for loop entry: assign loop pointers, counters
**/
LABEL( SUFFIX( aligned ) )
/**
 DST setup: bring in 2 cachelines
 MAKE STREAM_CODE( control_register, bytes_per_block, block_count,
 byte_stride )
**/
#if defined( DST_ENABLE )

#if defined( EXPAND_NCC )
   MR( dst rptr, Ar )
   MR( dst iptr, Ai )
#elif defined( EXPAND_CNC )
   MR( dst rptr, Br )
   MR( dst_iptr, Bi )
#endif

   MAKE STREAM CODE( dst control, 64, 1, 0 )
   DSTA( dst rptr, dst control )
   DSTA( dst iptr, dst control )
   DSTB( dst rptr, dst control )
   DSTB( dst_iptr, dst_control )
#endif

   SRWI C( count, N, NSHIFT )         /* 16 per trip */
   LI(addr incr, ADDRESS INCREMENT)  /* constants defined above */
   SLWI(ptr offset1, addr incr, 2)
   NEG(ptr_offset1, ptr_offset1)      /* will be adding addr_incr << 3 */

   ADD(Ar1, Ar0, addr incr)
   VXOR( rsumr, rsumr, rsumr )
   ADD(Br1, Br0, addr incr)
   ADD(Ai1, Ai0, addr incr)
   VXOR( rsumi, rsumi, rsumi )
   ADD(Bi1, Bi0, addr_incr)

   ADD(Ar2, Ar1, addr incr)
   VXOR( isumr, isumr, isumr )
   ADD(Br2, Br1, addr incr)
   ADD(Ai2, Ai1, addr incr)
   VXOR( isumi, isumi, isumi )
   ADD(Bi2, Bi1, addr_incr)
```

```
   ADD(Ar3, Ar2, addr incr)
   ADD(Br3, Br2, addr incr)
   ADD(Ai3, Ai2, addr incr)
   ADD(Bi3, Bi2, addr incr)
   SLWI(addr_incr, addr_incr, 3) /* bump by 8 elements */
/**
 Loop entry code
**/
   DSTA( dst rptr, dst control )
   LOAD A( ar0, Ar0, ptr offset0 )
   DSTB( dst rptr, dst control )
   LOAD B( br0, Br0, ptr offset0 )
   LOAD A( ai0, Ai0, ptr offset0 )
   LOAD_B( bi0, Bi0, ptr_offset0 )
/**
 Top of double loop structure
**/
LABEL( SUFFIX(loop0 ) )
    LOAD A( ar1, Ar1, ptr offset0 )
    VMADDFP( rsumr, ar0, br0, rsumr )
    DSTA( dst iptr, dst control )
    LOAD B( br1, Br1, ptr offset0 )
    VMADDFP( rsumi, ai0, bi0, rsumi )
    LOAD A( ai1, Ai1, ptr offset0 )
    LOAD B( bi1, Bi1, ptr offset0 )
    DSTB( dst iptr, dst_control )
    DECR C( count )
    LOAD A( ar2, Ar2, ptr offset0 )
    VMADDFP( isumi, ar0, bi0, isumi )
    VMADDFP( isumr, ai0, br0, isumr )
    LOAD B( br2, Br2, ptr offset0 )
    VMADDFP( rsumr, ar1, br1, rsumr )
    ADD(ptr offset1, ptr_offset1, addr_incr)
    VMADDFP( rsumi, ai1, bi1, rsumi )
    LOAD A( ai2, Ai2, ptr offset0 )
    VMADDFP( isumi, ar1, bi1, isumi )
    LOAD B( bi2, Bi2, ptr offset0 )
    VMADDFP( isumr, ai1, br1, isumr )
    VMADDFP( rsumr, ar2, br2, rsumr )
    LOAD A( ar3, Ar3, ptr offset0 )
    VMADDFP( rsumi, ai2, bi2, rsumi )
    LOAD B( br3, Br3, ptr offset0 )
    LOAD A( ai3, Ai3, ptr offset0 )
    VMADDFP( isumi, ar2, bi2, isumi )
    LOAD B( bi3, Bi3, ptr offset0 )
    VMADDFP( isumr, ai2, br2, isumr )
    BEQ( SUFFIX(loop0 exit ) )
    DSTA( dst rptr, dst control )
    LOAD A( ar0, Ar0, ptr offset1 )
    VMADDFP( rsumr, ar3, br3, rsumr )
    VMADDFP( rsumi, ai3, bi3, rsumi )
    DSTB( dst rptr, dst control )
    LOAD B( br0, Br0, ptr offset1 )
    VMADDFP( isumi, ar3, bi3, isumi )
    LOAD A( ai0, Ai0, ptr offset1 )
    LOAD B( bi0, Bi0, ptr offset1 )
    VMADDFP( isumr, ai3, br3, isumr )
    BR( SUFFIX(loop1 ) )
/**
 loop exit
**/
LABEL( SUFFIX(loop0 exit ) )
   MR(ptr offset0, ptr offset1)
   BR( SUFFIX(loop1_exit ) )
/**
 Top of second loop
```

```
**/
LABEL( SUFFIX(loop1 ) )
    LOAD A( ar1, Ar1, ptr offset1 )
    VMADDFP( rsumr, ar0, br0, rsumr )
    DSTA( dst iptr, dst control )
    LOAD B( br1, Br1, ptr offset1 )
    VMADDFP( rsumi, ai0, bi0, rsumi )
    LOAD A( ai1, Ai1, ptr offset1 )
    LOAD B( bi1, Bi1, ptr offset1 )
    DSTB( dst iptr, dst_control )
    DECR C( count )
    LOAD A( ar2, Ar2, ptr offset1 )
    VMADDFP( isumi, ar0, bi0, isumi )
    VMADDFP( isumr, ai0, br0, isumr )
    LOAD B( br2, Br2, ptr offset1 )
    VMADDFP( rsumr, ar1, br1, rsumr )
    ADD(ptr offset0, ptr_offset0, addr_incr)
    VMADDFP( rsumi, ai1, bi1, rsumi )
    LOAD A( ai2, Ai2, ptr offset1 )
    VMADDFP( isumi, ar1, bi1, isumi )
    LOAD B( bi2, Bi2, ptr offset1 )
    VMADDFP( isumr, ai1, br1, isumr )
    VMADDFP( rsumr, ar2, br2, rsumr )
    LOAD A( ar3, Ar3, ptr offset1 )
    VMADDFP( rsumi, ai2, bi2, rsumi )
    LOAD B( br3, Br3, ptr offset1 )
    LOAD A( ai3, Ai3, ptr offset1 )
    VMADDFP( isumi, ar2, bi2, isumi )
    LOAD B( bi3, Bi3, ptr offset1 )
    VMADDFP( isumr, ai2, br2, isumr )
    BEQ( SUFFIX(loop1 exit ) )
    DSTA( dst rptr, dst control )
    LOAD A( ar0, Ar0, ptr offset0 )
    VMADDFP( rsumr, ar3, br3, rsumr )
    VMADDFP( rsumi, ai3, bi3, rsumi )
    DSTB( dst rptr, dst control )
    LOAD B( br0, Br0, ptr offset0 )
    VMADDFP( isumi, ar3, bi3, isumi )
    LOAD A( ai0, Ai0, ptr offset0 )
    LOAD B( bi0, Bi0, ptr offset0 )
    VMADDFP( isumr, ai3, br3, isumr )
    BR( SUFFIX(loop0 ) )
/**
 Drop out of loop, flush pipe
**/
LABEL( SUFFIX(loop1 exit ) )
   VMADDFP( rsumr, ar3, br3, rsumr )
   VMADDFP( rsumi, ai3, bi3, rsumi )
   VMADDFP( isumi, ar3, bi3, isumi )
   VMADDFP( isumr, ai3, br3, isumr )
/**
 Remaining sum updates
**/
LABEL( SUFFIX(two_left) )
   ANDI_C( count, N, 0x8 )  /* bit 3 */
   BEQ( SUFFIX(one_left ) )

   LOAD A( ar0, Ar0, ptr offset0 )
   LOAD B( br0, Br0, ptr offset0 )
   LOAD A( ai0, Ai0, ptr offset0 )
   LOAD_B( bi0, Bi0, ptr_offset0 )

   LOAD A( ar1, Ar1, ptr offset0 )
   LOAD B( br1, Br1, ptr offset0 )
   LOAD A( ai1, Ai1, ptr offset0 )
   LOAD_B( bi1, Bi1, ptr_offset0 )
```

```
    VMADDFP( rsumr, ar0, br0, rsumr )
    VMADDFP( rsumi, ai0, bi0, rsumi )
    VMADDFP( isumi, ar0, bi0, isumi )
    VMADDFP( isumr, ai0, br0, isumr )

    VMADDFP( rsumr, ar1, br1, rsumr )
    VMADDFP( rsumi, ai1, bi1, rsumi )
    VMADDFP( isumi, ar1, bi1, isumi )
    VMADDFP( isumr, ai1, br1, isumr )
    ADDI( ptr_offset0, ptr_offset0, 32 )

LABEL( SUFFIX(one_left) )
    ANDI_C( count, N, 0x4 )  /* bit 2 */
    BEQ( SUFFIX(combine ) )
    LOAD_A( ar0, Ar0, ptr_offset0 )
    LOAD_B( br0, Br0, ptr_offset0 )
    LOAD_A( ai0, Ai0, ptr_offset0 )
    LOAD_B( bi0, Bi0, ptr_offset0 )
    VMADDFP( rsumr, ar0, br0, rsumr )
    VMADDFP( rsumi, ai0, bi0, rsumi )
    VMADDFP( isumi, ar0, bi0, isumi )
    VMADDFP( isumr, ai0, br0, isumr )
    ADDI( ptr_offset0, ptr_offset0, 16 )
/**
 combine partial sums, permute, write out results
**/
LABEL( SUFFIX(combine) )
  VSUBFP( rsumr, rsumr, rsumi ) /* rsumr = rsumr - rsumi */
  VADDFP( isumi, isumi, isumr )
/**
 8 bytes/cycle shuffle:
 real/imag logic should be intermixed for efficiency
**/
  VMRGHW(rsum0, rsumr, rsumr)
  ANDI_C( addr_incr, N, 0x3 )
  VMRGHW(isum0, isumi, isumi)
  VMRGLW(rsum1, rsumr, rsumr)
  SUB( addr_incr, N, addr_incr ) /* offset index for remainders */
  VMRGLW(isum1, isumi, isumi)
  VADDFP( rsum0, rsum1, rsum0 )
  SLWI(addr_incr, addr_incr, 2) /* byte offset */
  VADDFP( isum0, isum1, isum0 )

  VMRGHW(rsum1, rsum0, rsum0)
  ADD(Ar0, Ar0, addr_incr)
  VMRGHW(isum1, isum0, isum0)
  ADD(Ai0, Ai0, addr_incr)
  VMRGLW(rsum0, rsum0, rsum0)
  ADD(Br0, Br0, addr_incr)
  VMRGLW(isum0, isum0, isum0)
  ADD(Bi0, Bi0, addr_incr)
  VADDFP( rsumr, rsum1, rsum0 )
  LI(ptr_offset0, 0) /* needed for output */
  VADDFP( isumi, isum1, isum0 )
/**
 4 byte stores
**/
  STVEWX( rsumr, Cr, ptr_offset0 )
  STVEWX( isumi, Ci, ptr_offset0 )
/**
 Remainders of 1-3 more to do
**/
  ANDI_C( N, N, 3 )
  LFS( rsum_vmx, Cr, 0 )
  LFS( isum_vmx, Ci, 0 )
  BEQ( SUFFIX( scaler_vmx_combine ) )
```

zdotpr_vmx.k 2/23/2001

```
/**
 Here to do last 1-3 points using standard FP
**/
LABEL( SUFFIX( post_loop ) )
   LFS( far, Ar0, 0 )
   LFS( fai, Ai0, 0 )
   DECR_C( N )
   LFS( fbr, Br0, 0 )
   LFS( fbi, Bi0, 0 )
   FMADDS( frsumr, far, fbr, frsumr )
   FMADDS( frsumi, fai, fbi, frsumi )
   FMADDS( fisumi, far, fbi, fisumi )
   FMADDS( fisumr, fai, fbr, fisumr )
   ADDI(Ar0, Ar0, 4)
   ADDI(Br0, Br0, 4)
   ADDI(Ai0, Ai0, 4)
   ADDI(Bi0, Bi0, 4)
   BNE( SUFFIX( post_loop) )
/**
 Write out result
**/
LABEL( SUFFIX( scaler vmx combine ) )
  FSUBS( frsum, frsumr, frsumi ) /* rsumr = rsumr - rsumi */
  FADDS( fisum, fisumi, fisumr )
  FADDS( frsum, frsum, rsum vmx )
  FADDS( fisum, fisum, isum_vmx )
  STFS( frsum, Cr, 0 )
  STFS( fisum, Ci, 0 )
/**
 return
**/
LABEL( SUFFIX(ret) )
   FREE THRU v19( VREGSAVE_COND )
   REST r13_r30
   RETURN
FUNC_EPILOG
```

```
zdotpr_vmx.mac                                                    2/23/2001

#define ZDOTPR 0
#define ZIDOTPR 1

/*-------------------------------------------------------------------+
|--- MC Standard Algorithms -- PPC Macro language Version        ---|
|-------------------------------------------------------------------|
|                                                                   |
|  File Name:     ZDOTPR.MAC                                        |
|  Description:   Vector Single Precision Complex Dot Product       |
|  Entry/params:  ZDOTPR (A, I, B, J, C, N)                         |
|  Formula: C[0] = sum (A->realp[mI]*B->realp[mJ]                   |
|                       - A->imagp[mI]*B->imagp[mJ])                |
|           C[1] = sum (A->realp[mI]*B->imagp[mJ]                   |
|                       + A->imagp[mI]*B->realp[mJ])                |
|                   for m=0 to N-1                                  |
|                                                                   |
|            Mercury Computer Systems, Inc.                         |
|            Copyright (c) 1998  All rights reserved                |
|                                                                   |
|   Revision      Date      Engineer Reason                         |
|   --------      ----      -------- --------                       |
|     0.0        981209       fpl   Created (from cdotpr.mac)       |
|     0.1        990310       fpl   750/G4 integration              |
|     0.1        990322       fpl   Stylistic changes               |
+-------------------------------------------------------------------*/

#define COMPILE_ESAL_JUMP_TABLE

#define FUNC_TYPE ZDOTPR

#if defined( BUILD_MAX )

#undef VMX SAL
#undef VMX NN
#undef VMX NC
#undef VMX CN
#undef VMX_CC

#if !defined( COMPILE ESAL_JUMP_TABLE ) || defined(
COMPILE_NO_ESAL_JUMP_TABLE )
                              /* 1 variant: _zdotpr_vmx() */
#define VMX SAL
#include "zdotpr_vmx.k"

#else                         /* 5 variants based on ESAL flag */

#define VMX NN
#include "zdotpr_vmx.k"

#undef VMX NN
#define VMX NC
#include "zdotpr_vmx.k"

#undef VMX NC
#define VMX CN
#include "zdotpr_vmx.k"

#undef VMX CN
#define VMX CC
#include "zdotpr_vmx.k"
#undef VMX_CC
#endif                        /* end COMPILE_ESAL_JUMP_TABLE */

#endif                        /* end BUILD_MAX */
```

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Implicit Waveform Subtraction

1. In a spread spectrum communication system of the type that processes one or more spread-spectrum waveforms ("user spread-spectrum waveforms"), each representative of a waveform associated with a respective user, the improvement comprising:

   a first logic element that generates a residual composite spread-spectrum waveform as a function of a composite spread-spectrum waveform and an estimated composite spread-spectrum waveform,

   one or more second logic elements each coupled to the first logic element, each second logic element generating a refined matched-filter detection statistic for at least a selected user as a function of

   (i) the residual composite spread-spectrum waveform and

   (ii) a characteristic of an estimate of the selected user's spread-spectrum waveform.

2. In the system of claim 1, the further improvement wherein the characteristic is at least one of an estimated amplitude and an estimated symbol associated with the estimate of the selected user's spread-spectrum waveform..

3. In the system of claim 1, the improvement wherein the spread-spectrum communications system comprises a code division multiple access (CDMA) base station.

4. In the system of claim 1, the improvement wherein the CDMA base station comprises one or more long-code receivers, and each long-code receiver generating one or more respective matched-filter detection statistics, from which the estimated composite spread-spectrum waveform is, in part, generated.

5. In the system of claim 1, the improvement wherein the first logic element comprises summation logic which generates the residual composite spread-spectrum waveform based on the relation

$$r_{res}^{(n)}[t] \equiv r[t] - \hat{r}^{(n)}[t],$$

wherein

$r_{res}^{(n)}[t]$ is the residual composite spread-spectrum waveform,

$r[t]$ represents the composite spread-spectrum waveform,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

t is a sample time period, and

n is an iteration count.

6. In the system of claim 5, the further improvement wherein the estimated composite spread-spectrum waveform is pulse-shaped and is based on estimated complex amplitudes, estimated delay lags, estimated symbols, and codes of the one or more user spread-spectrum waveforms.

7. In the system of claim 1, the further improvement wherein each second logic element comprises rake logic and summation logic which generates the refined matched-filter detection statistics based on the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_{res,k}^{(n)}[m]$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{res,k}^{(n)}[m]$ represents a residual matched-filter detection statistic for the k^{th} user, and

n is an iteration count.

8. In the system of claim 1, the further improvement wherein the refined matched-filter detection statistic for each user is iteratively generated.

9. In the system of claim 1, the further improvement wherein the refined matched-filter detection statistic for at least a selected user is generated by a long-code receiver.

10. In the system of claim 1, the improvement wherein the first and second logic elements are implemented on any of processors, field programmable gate arrays, array processors and co-processors, or any combination thereof.

11. In a spread spectrum communication system of the type that processes one or more user spread-spectrum waveforms, each representative of a waveform associated with a respective user, the improvement comprising:

a first logic element which generates an estimated composite spread-spectrum waveform that is a function of estimated user complex channel amplitudes, time lags, and user codes,

a second logic element coupled to the first logic element, the second logic element generating a residual composite spread-spectrum waveform a function of a composite user spread-spectrum waveform and the estimated composite spread-spectrum waveform,

one or more third logic elements each coupled to the second logic element, the third logic element generating a refined matched-filter detection statistic for at least a selected user as a function of

(i) the residual composite spread-spectrum waveform and

(ii) a characteristic of an estimate of the selected user's spread-spectrum waveform.

12. In the system of claim 11, the further improvement wherein the characteristic is at least one of an estimated amplitude, an estimated delay lag and an estimated symbol associated with the estimate of the selected user's spread-spectrum waveform.

13. In the system of claim 11, the improvement wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

14. In the system of claim 13, the improvement wherein the CDMA base station comprises long-code receivers.

15. In the system of claim 11, the improvement wherein the first logic element further comprises arithmetic logic which generates the estimated composite spread-spectrum waveform based on the relation

$$\hat{r}^{(n)}[t] = \sum_r g[r]\rho^{(n)}[t-r],$$

wherein

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

$g[t]$ represents a raised-cosine pulse shape.

16. In the system of claim 15, the further improvement wherein the first logic element comprises arithmetic logic which generates an estimated composite re-spread waveform based on the relation

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v}\sum_{p=1}^{L}\sum_r \delta[t-\hat{\tau}_{kp}^{(n)}-rN_c]\cdot\hat{a}_{kp}^{(n)}\cdot c_k[r]\cdot\hat{b}_k^{(n)}[\lfloor r/N_k \rfloor],$$

wherein

K_v is a number of simultaneous dedicated physical channels for all users,

$\delta[t]$ is a discrete-time delta function,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

$c_k[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th}th multipath component for the k^{th} user,

N_k is a spreading factor for the k^{th} user,

t is a sample time index,

L is a number of multi-path components.,

N_c is a number of samples per chip, and

n is an iteration count.

17. In the system of claim 11, the improvement wherein the second logic element comprises summation logic which generates the residual composite spread-spectrum waveform that based on the relation

$$r_{res}^{(n)}[t] \equiv r[t] - \hat{r}^{(n)}[t],$$

wherein

$r_{res}^{(n)}[t]$ is the residual composite spread-spectrum waveform ,

$r[t]$ represents the composite spread-spectrum waveform,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

t is a sample time period, and

n is an iteration count.

18. In the system of claim 17, the further improvement wherein the estimated composite spread-spectrum waveform is pulse-shaped and is based on the user spread-spectrum waveform.

19. In the system of claim 18, the further improvement wherein each third logic element comprises rake logic and summation logic which generates the second user matched-filter detection statistic based on the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_{res,k}^{(n)}[m],$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{res,k}^{(n)}[m]$ represents the user residual matched-filter detection statistic for the m^{th} symbol period, and

n is an iteration count.

20. In the system of claim 11, the further improvement wherein the refined matched-filter detection statistic for each user is iteratively generated.

21. In the system of claim 11, the improvement wherein the logic elements are implemented on any of a processors, field programmable gate arrays, array processors and co-processors, or any combination thereof.

22. A method for multiple user detection in a spread-spectrum communication system that processes long-code spread-spectrum user transmitted waveforms comprising:

generating a residual composite spread-spectrum waveform as a function of an arithmetic difference between a composite spread-spectrum waveform and an estimated spread-spectrum waveform,

generating a refined matched-filter detection statistic that is a function of a sum of a rake-processed residual composite spread-spectrum waveform for a selected user and an amplitude statistic for that selected user.

23. The method of claim 22, comprising generating a refined matched-filter detection statistic that is a function of a sum of a rake-processed residual composite spread-spectrum waveform for a selected user and an amplitude statistic for that selected user multiplied by a soft symbol estimate.

24. The method of claim 22, further wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

25. The method of claim 22, wherein the step of generating the residual composite spread-spectrum waveform further comprises performing arithmetic logic that is based on the relation

$$r_{res}^{(n)}[t] \equiv r[t] - \hat{r}^{(n)}[t],$$

wherein

$r_{res}^{(n)}[t]$ is the residual composite spread-spectrum waveform ,

$r[t]$ represents the composite spread-spectrum waveform,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

t is a sample time period, and

n is an iteration count.

26. The method of claim 22, wherein the estimated composite spread-spectrum waveform is pulse-shaped and is based on a composite user re-spread waveform.

27. The method of claim 22, wherein the step of generating the refined matched-filter detection statistic representative of that user further comprises performing arithmetic logic based on the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_{res,k}^{(n)}[m]$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{res,k}^{(n)}[m]$ represents a residual matched-filter detection statistic, and

n is an iteration count.

28. The method of claim 22, the further improvement wherein the refined matched-filter detection statistic is generated by a long-code receiver.

29. The method of claim 22, the further improvement wherein the step of generating the residual matched-filter detection statistic for an m^{th} symbol period comprises performing arithmetic logic based on the relation

$$y_{res,k}^{(n)}[m] \equiv \mathrm{Re}\left\{\sum_{p=1}^{L} \hat{a}_{kp}^{(n)H} \cdot \frac{1}{2N_k} \sum_{r=0}^{N_k-1} r_{res}^{(n)}[rN_c + \hat{\tau}_{kp}^{(n)} + mT_k] \cdot c_{km}^{*}[r]\right\}$$

wherein

$y_{res,k}^{(n)}[m]$ represents the user residual matched-filter detection statistic for the m^{th} symbol period,

L is a number of multi-path components,

$\hat{a}_{kp}^{(n)}$ is the estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

N_k is the spreading factor for the k^{th} user,

$r_{res}^{(n)}[t]$ is the residual composite spread-spectrum waveform ,

N_c is the number of samples per chip, and

$\hat{\tau}_{kp}^{(n)}$ is the time lag for the p^{th} multipath component for the k^{th} user ,

m is a symbol period,

T_k is a channel symbol duration for the k^{th} user,

$c_{km}[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels.

n is an iteration count.

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Matched-filter Outputs

30. In a spread spectrum communication system of the type that processes one or more spread-spectrum waveforms ("user spread-spectrum waveforms"), each representative of a waveform associated with a respective user, the improvement comprising:

a first logic element which generates an estimated composite spread-spectrum waveform that is a function of one or more of estimated complex amplitudes, estimated time lags, estimated symbols, and codes of the one or more user spread-spectrum waveforms,

one or more second logic elements each coupled to the first logic element, the one or more second logic elements generating a second matched-filter detection statistic for at least a selected user as a function of a difference between a first matched-filter detection statistic for that user and an estimated matched-filter detection statistic for that user as a function of the estimated composite spread-spectrum waveform.

31. In the system of claim 30, the further improvement wherein the one or more second logic elements generate the second matched-filter detection statistic for at least the selected user as a function of a difference between (i) a sum of the first matched-filter detection statistic for that user and a characteristic of an estimate of the selected user's spread-spectrum waveform and (ii) the estimated matched-filter detection statistic for that user.

32. In the system of claim 31, the further improvement wherein the characteristic is at least one of an estimated amplitude and an estimated symbol associated with the estimate of the selected user's spread-spectrum waveform.

33. In the system of claim 30, the improvement wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

34. In the system of claim 33, the improvement wherein the CDMA base station comprises long-code receivers.

35. In the system of claim 30, the improvement further wherein the first logic element comprises arithmetic logic which generates an estimated composite re-spread waveform based on the relation

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v} \sum_{p=1}^{L} \sum_{r} \delta[t - \hat{\tau}_{kp}^{(n)} - rN_c] \cdot \hat{a}_{kp}^{(n)} \cdot c_k[r] \cdot \hat{b}_k^{(n)}[\lfloor r / N_k \rfloor],$$

wherein

K_v is a number of simultaneous dedicated physical channels for all users,

$\delta[t]$ is a discrete-time delta function,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

$c_k[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user,

N_k is a spreading factor for the k^{th} user,

t is a sample time index,

L is a number of multi-path components.,

N_c is a number of samples per chip, and

n is an iteration count.

36. In the system of claim 35, the improvement wherein the first logic element further comprises arithmetic logic which generates the estimated composite spread-spectrum waveform based on the relation

$$\hat{r}^{(n)}[t] = \sum_{r} g[r]\rho^{(n)}[t-r],$$

wherein

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform, and

$g[t]$ represents a pulse shape.

37. In the system of claim 30, the improvement further wherein the estimated composite residual spread-spectrum waveform is pulse-shaped and is based on the user spread-spectrum waveform.

38. In the system of claim 30, the improvement further wherein each second logic element comprises rake logic and summation logic which generates the second matched-filter detection statistic based on the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_k^{(n)}[m] - y_{est,k}^{(n)}[m],$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the mth symbol period ,

$y_k^{(n)}[m]$ represents the first matched-filter detection statistic for the selected user,

$y_{est,k}^{(n)}[m]$ represents the estimated matched-filter detection statistic for the selected user, and

n is an iteration count.

39. In the system of claim 38, the improvement further wherein the system generates the second matched-filter detection statistic for the selected user and zero, one or more further second matched-filter detection statistics for that user iteratively.

40. In the system of claim 30, the improvement further wherein the first matched-filter detection statistic for at least the selected user is generated by a long-code receiver.

41. In the system of claim 30, the improvement wherein the logic elements are implemented on any of a processors, field programmable gate arrays, array processors and co-processors, or any combination thereof.

42. In a method for multiple user detection in a spread-spectrum communication system that processes long-code spread-spectrum user waveforms, the improvement comprising a method of generating user matched-filter detection statistics for at least a selected user comprising:

generating a composite spread-spectrum waveform as a function of a pulsed-shaped composite re-spread waveform,

generating a refined user matched-filter detection statistic for at least the selected user that is a function of a difference between a first matched-filter detection statistic for that user and an estimated matched-filter detection statistic for that user.

43. In the method of claim 42, the further improvement comprising generating the refined-matched-filter detection statistic for at least the selected user as a function of a difference between (i) the sum of the first matched-filter detection statistic for that user and a characteristic of an estimate of the selected user's spread-spectrum waveform and (ii) the estimated matched-filter detection statistic for that user.

44. In the method of claim 43, the further improvement wherein the characteristic is at least one of an estimated amplitude, and an estimated symbol associated with an estimate of the selected user's spread-spectrum waveform.

45. In the method of claim 42, further wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

46. In the method of claim 42, wherein the step of generating the composite spread-spectrum waveform further comprises a function of a composite signal representing the sum of all estimated user waveforms.

47. In the method of claim 46, the improvement further wherein the function of the composite signal representing the sum of all estimated user waveforms comprises a pulse-shaping filter.

48. In the method of claim 42, wherein the step of generating the second matched-filter detection statistic representative of that user further comprises performing arithmetic logic based on the relation

$$y_k^{(n+1)}[m] = A_k^{(n)2} \cdot \hat{b}_k^{(n)}[m] + y_k^{(n)}[m] - y_{est,k}^{(n)}[m],$$

wherein

$A_k^{(n)2}$ represents an amplitude statistic,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the mth symbol period,

$y_k^{(n)}[m]$ represents the first matched-filter detection statistic,

$y_{est,k}^{(n)}[m]$ represents the estimated matched-filter detection statistic, and

n is an iteration count.

49. In the method of claim 48, the further improvement wherein second matched-filter detection statistic is derived from the estimated composite spread-spectrum waveform based on the relation

$$y_{est,k}^{(n)}[m] \equiv \mathrm{Re}\left\{ \sum_{p=1}^{L} \hat{a}_{kp}^{(n)H} \cdot \frac{1}{2N_k} \sum_{r=0}^{N_k-1} \hat{r}^{(n)}[rN_c + \hat{\tau}_{kp}^{(n)} + mT_k] \cdot c_{km}^{*}[r] \right\},$$

wherein

L is a number of multi-path components,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

N_k is a spreading factor for the k^{th} user,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

N_c is a number of samples per chip, and

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user,

m is a symbol period,

T_k is a data bit duration,

n is an iteration count and

$c_{km}[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels.

Wireless Communication Systems And Methods For Long-code Communications For Regenerative Multiple User Detection Involving Pre-maximal Combination Matched Filter Outputs

50. In a spread spectrum communication system of the type that processes one or more spread-spectrum waveforms ("user spread-spectrum waveforms"), each representative of a waveform received from a respective user, the improvement comprising:

one or more first logic elements generating a first complex channel amplitude estimate corresponding to at least a selected user and at least a selected finger of a rake receiver that receives the selected user waveform.

one or more second logic elements each coupled to one or more first logic elements, each generating an estimated composite spread-spectrum waveform that is a function of one or more of estimated complex channel amplitudes, estimated delay lags, estimated symbols, and/or codes of the one or more user spread-spectrum waveforms,

one or more third logic elements each coupled to one or more second logic elements, the one or more third logic elements generating a second pre-combination matched-filter detection statistic for at least a selected user and for at least a selected finger as a

function of a first pre-combination matched-filter detection statistic for that user and a pre-combination estimated matched-filter detection statistic for that user.

51. In the system of claim 50, the further improvement comprising

one or more fourth logic elements, each coupled to one or more third logic elements, the fourth logic element generating a second complex channel amplitude estimate corresponding to at least a selected user and at least selected finger.

52. In the system of claim 50, the further improvement wherein the one or more third logic elements generate the second pre-combination matched-filter detection statistic for at least the selected user and at least the selected finger as a function of a difference between (i) the sum of the first pre-combination matched-filter detection statistic for that user and that finger and a characteristic of an estimate of the selected user's spread-spectrum waveform and (ii) the pre-combination estimated matched-filter detection statistic for that user and that finger.

53. In the system of claim 52, the further improvement wherein the characteristic is at least one of an estimated amplitude and an estimated symbol associated with the estimate of the selected user's spread-spectrum waveform.

54. In the system of claim 50, the improvement wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

55. In the system of claim 54, the improvement wherein the CDMA base station comprises long-code receivers.

56. In the system of claim 50, the improvement further wherein the first and fourth logic elements comprise arithmetic logic which generate a complex channel amplitude estimate corresponding to at least a selected user and at least a selected finger of a rake receiver that receives the selected user waveform based on the relation

$$\hat{a}_{kp}^{(n)} \equiv \sum_{s} w[s] \cdot \frac{1}{N_p} \sum_{m=0}^{N_p-1} y_{kp}^{(n)}[m+Ms] \cdot b_k^{(n)}[m+Ms],$$

wherein

$\hat{a}_{kp}^{(n)}$ is a complex channel amplitude estimate corresponding to the p^{th} finger of the k^{th} user,

$w[s]$ is a filter,

N_p is a number of symbols,

$y_{kp}^{(n)}[m]$ is a first pre-combination matched-filter detection statistic corresponding to the p^{th} finger of the k^{th} user for the m^{th} symbol period,

M is a number of symbols per slot,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

m is a number symbol period index,

s is a slot index, and

n is an iteration count.

57. In the system of claim 50, the improvement further wherein the second logic element comprises arithmetic logic which generates an estimated composite re-spread waveform based on the relation

$$\rho^{(n)}[t] = \sum_{k=1}^{K_v} \sum_{p=1}^{L} \sum_{r} \delta[t - \hat{\tau}_{kp}^{(n)} - rN_c] \cdot \hat{a}_{kp}^{(n)} \cdot c_k[r] \cdot \hat{b}_k^{(n)}[\lfloor r / N_k \rfloor],$$

wherein

K_v is a number of simultaneous dedicated physical channels for all users,

$\delta[t]$ is a discrete-time delta function,

$\hat{a}_{kp}^{(n)}$ is an estimated complex channel amplitude for the p^{th} multipath component for the k^{th} user,

$c_k[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user ,

N_k is a spreading factor for the k^{th} user,

t is a sample time index,

L is a number of multi-path components.,

N_c is a number of samples per chip, and

n is an iteration count.

58. In the system of claim 57, the improvement wherein the second logic element further comprises arithmetic logic which generates the estimated composite spread-spectrum waveform based on the relation

$$\hat{r}^{(n)}[t] = \sum_{r} g[r]\rho^{(n)}[t-r],$$

wherein

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

$g[t]$ represents a pulse shape.

59. In the system of claim 50, the improvement further wherein the estimated composite residual spread-spectrum waveform is pulse-shaped and is based on the user spread-spectrum waveform.

60. In the system of claim 50, the improvement further wherein each third logic element comprises rake logic and summation logic which generates the second pre-combination matched-filter detection statistic based on the relation

$$y_{kp}^{(n+1)}[m] \equiv \hat{a}_{kp}^{(n)} \cdot \hat{b}_k^{(n)}[m] + y_{kp}^{(n)}[m] - y_{est,kp}^{(n)}[m],$$

wherein

$y_{kp}^{(n+1)}[m]$ represents the pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$\hat{a}_{kp}^{(n)}$ is the complex channel amplitude for the p^{th} finger for the k^{th} user,

$\hat{b}_{k}^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{kp}^{(n)}[m]$ represents the first pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$y_{est,kp}^{(n)}[m]$ represents the pre-combination estimated matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period, and

n is an iteration count.

61. In the system of claim 60, the improvement further wherein the system generates the second pre-combination matched-filter detection statistic for the selected user and finger and zero, one or more further matched-filter detection statistics for that user and finger iteratively.

62. In the system of claim 55A, the improvement further wherein the system generates the second complex channel amplitude estimates for the selected user and finger and zero, one or more further complex channel amplitude estimates for that user and finger iteratively.

63. In the system of claim 50, the improvement further wherein the first pre-combination matched-filter detection statistic for at least the selected user and finger is generated by a long-code receiver.

64. In the system of claim 50, the improvement wherein the logic elements are implemented on any of a processors, field programmable gate arrays, array processors and co-processors, or any combination thereof.

65. In a method for multiple user detection in a spread-spectrum communication system that processes long-code spread-spectrum user waveforms, the improvement comprising a method of generating user pre-combination matched-filter detection statistics for at least a selected user and finger comprising:

generating a composite spread-spectrum waveform as a function of a pulsed-shaped composite re-spread waveform,

generating a second user pre-combination matched-filter detection statistic for at least the selected user and finger that is a function of a difference between a first pre-combination matched-filter detection statistic for that user and finger and a pre-combination estimated matched-filter detection statistic for that user and finger.

66. In the method of claim 65, the further improvement comprising generating the second pre-combination matched-filter detection statistic for at least the selected user and finger as a function of a difference between (i) the sum of the first pre-combination matched-filter detection statistic for that user and finger and a characteristic of an estimate of the selected user's spread-spectrum waveform and (ii) the pre-combination estimated matched-filter detection statistic for that user.

67. In the method of claim 66, the further improvement wherein the characteristic is at least one of an estimated amplitude, and an estimated symbol associated with an estimate of the selected user's spread-spectrum waveform.

68. In the method of claim 65, further wherein the spread-spectrum communications system is a code division multiple access (CDMA) base station.

69. In the method of claim 65, wherein the step of generating the composite spread-spectrum waveform further comprises a function of a composite signal representing the sum of all estimated user waveforms.

70. In the method of claim 69, the improvement further wherein the function of the composite signal representing the sum of all estimated user waveforms comprises a pulse-shaping filter.

71. In the method of claim 16, wherein the step of generating the second pre-combination matched-filter detection statistic representative of that user and finger further comprises performing arithmetic logic based on the relation

$$y_{kp}^{(n+1)}[m] \equiv \hat{a}_{kp}^{(n)} \cdot \hat{b}_k^{(n)}[m] + y_{kp}^{(n)}[m] - y_{est,kp}^{(n)}[m],$$

wherein

$y_{kp}^{(n+1)}[m]$ represents the pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$\hat{a}_{kp}^{(n)}$ is the complex channel amplitude for the p^{th} finger for the k^{th} user,

$\hat{b}_k^{(n)}[m]$ represents a soft symbol estimate for the k^{th} user for the m^{th} symbol period,

$y_{kp}^{(n)}[m]$ represents the first pre-combination matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period,

$y_{est,kp}^{(n)}[m]$ represents the pre-combination estimated matched-filter detection statistic for the p^{th} finger for the k^{th} user for the m^{th} symbol period, and

72. In the method of claim 71, the further improvement wherein second pre-combination matched-filter detection statistic is derived from the estimated composite spread-spectrum waveform based on the relation

$$y_{est,kp}^{(n)}[m] \equiv \frac{1}{2N_k} \sum_{r=0}^{N_k-1} \hat{r}^{(n)}[rN_c + \hat{\tau}_{kp}^{(n)} + mT_k] \cdot c_{km}^{*}[r],$$

wherein

N_k is a spreading factor for the k^{th} user,

$\hat{r}^{(n)}[t]$ represents the estimated composite spread-spectrum waveform,

N_c is a number of samples per chip, and

$\hat{\tau}_{kp}^{(n)}$ is an estimated time lag for the p^{th} multipath component for the k^{th} user,

m is a symbol period,

T_k is a data bit duration, and

n is an iteration count.

$c_{km}[r]$ represents a user code comprising at least a scrambling code, an orthogonal variable spreading factor code, and a j factor associated with even numbered dedicated physical channels.

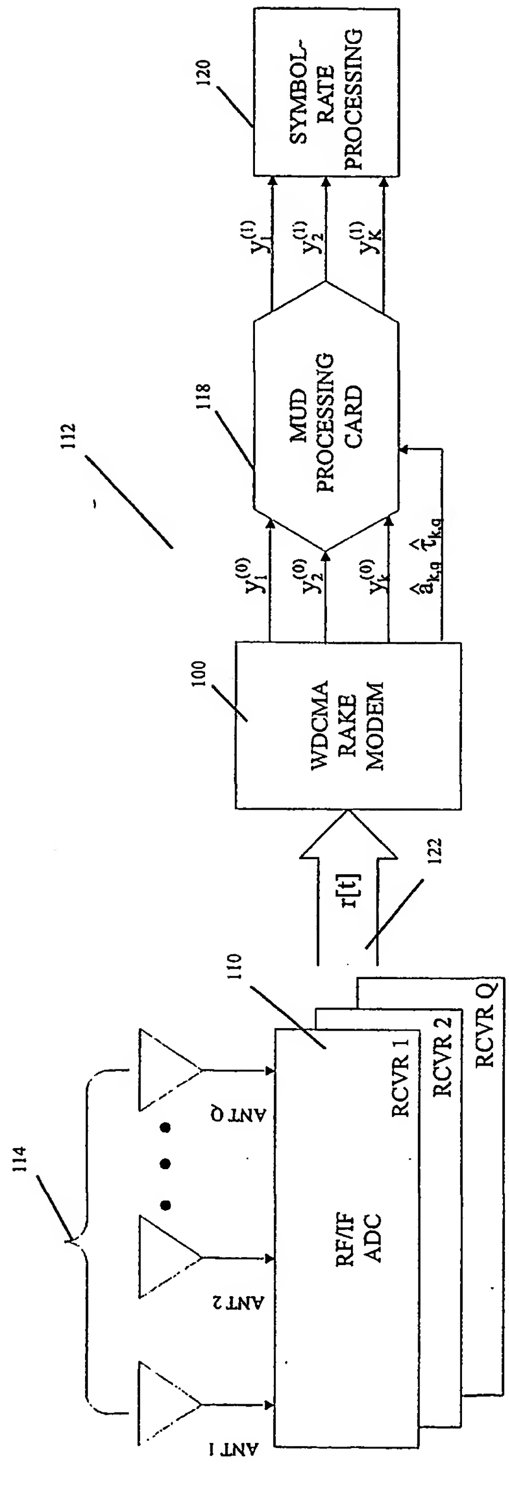

120
SYMBOL-RATE PROCESSING
118
MUD PROCESSING CARD
112
100
WDCMA RAKE MODEM
r[t]
122
110
RF/IF ADC
RCVR 1
RCVR 2
RCVR Q
114
ANT 1
ANT 2
ANT Q

Figure 1

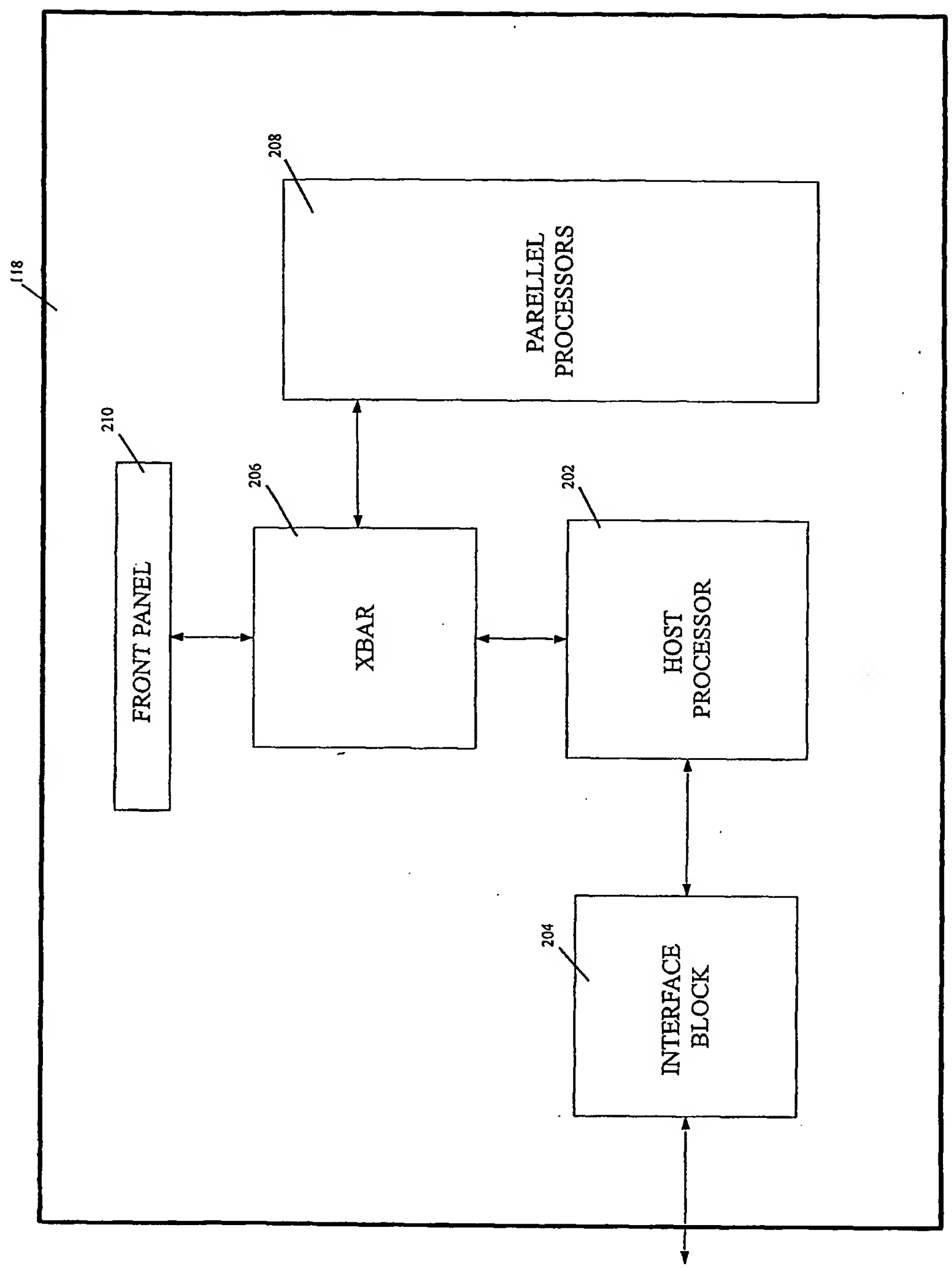

118
208
PARELLEL PROCESSORS
210
FRONT PANEL
206
XBAR
202
HOST PROCESSOR
204
INTERFACE BLOCK

Figure 2

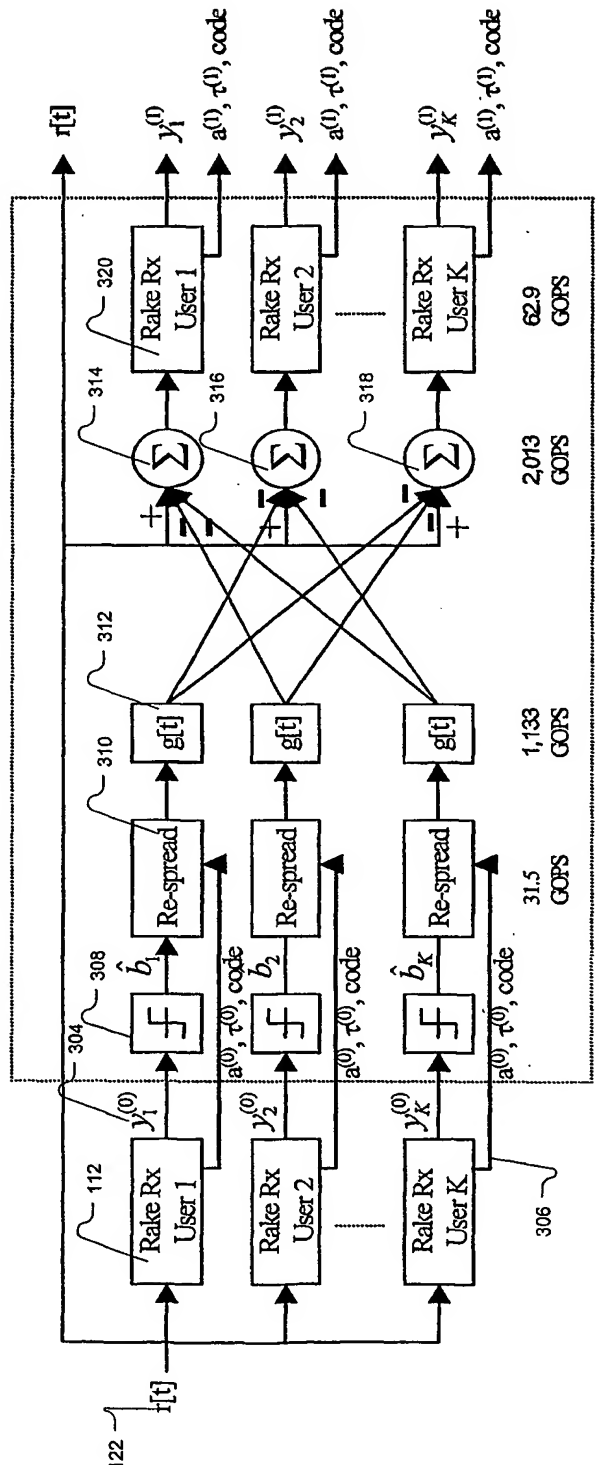

r[t]
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
Re-spread
g[t]
31.5 GOPS
1,133 GOPS
2,013 GOPS
62.9 GOPS
122
112
304
306
308
310
312
314
316
318
320

Figure 3

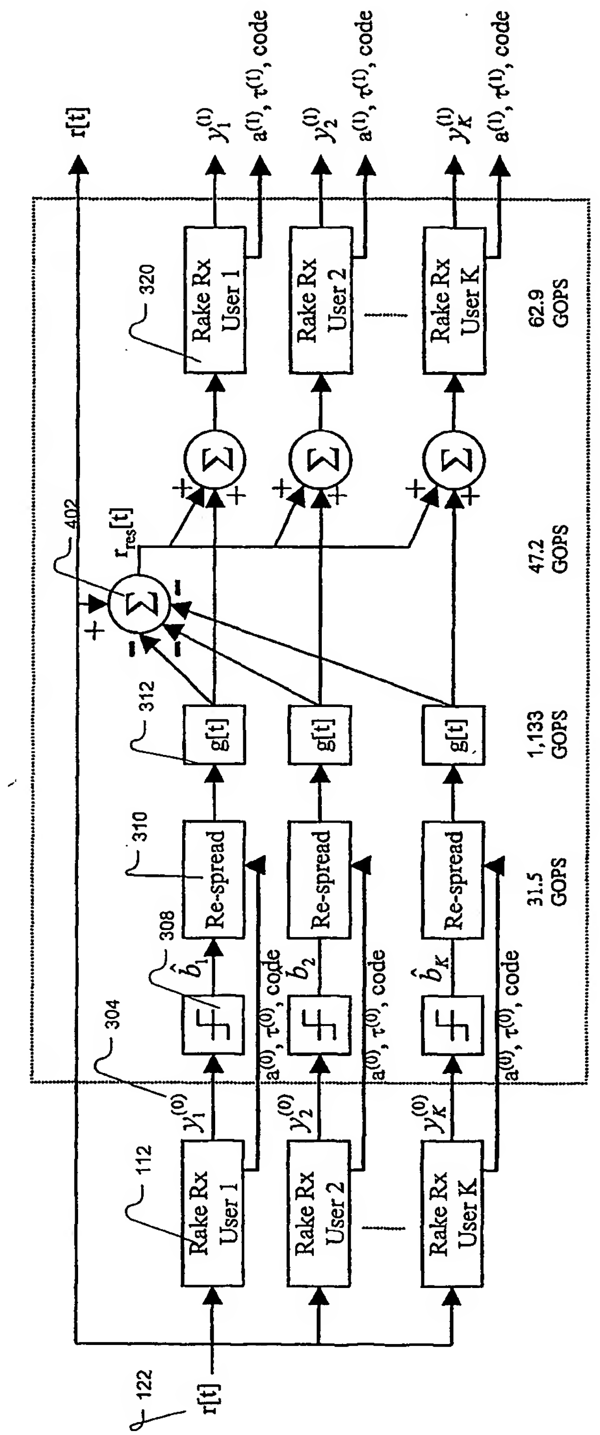
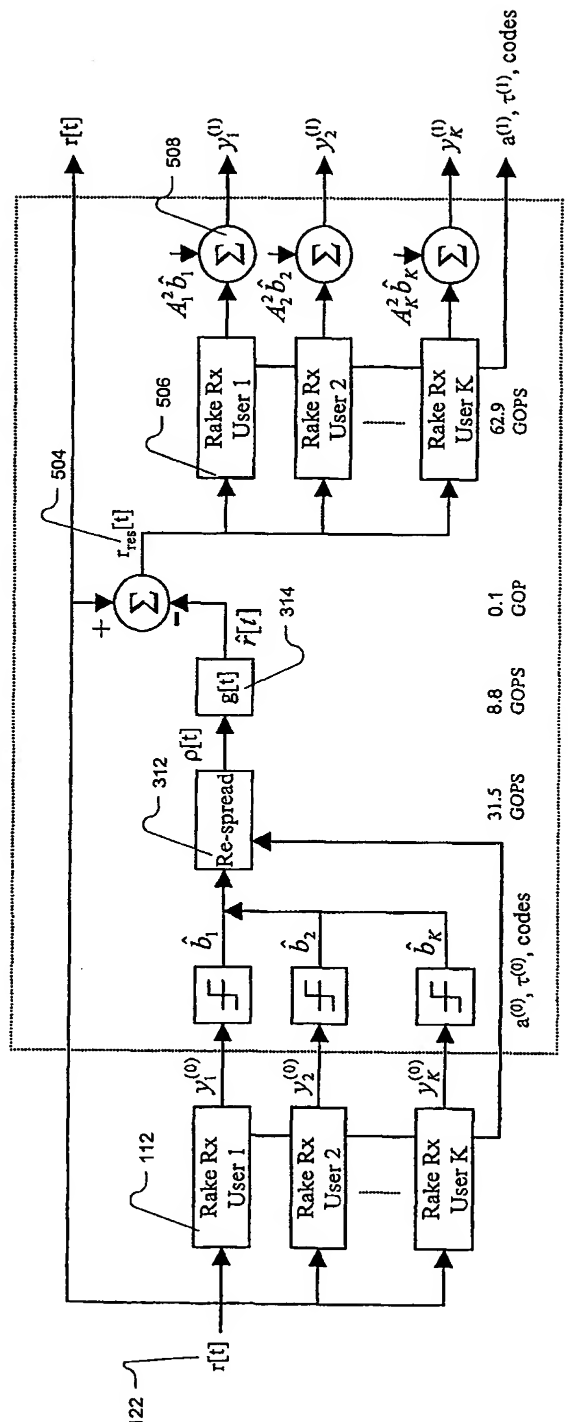

r[t]
122
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
112
$y_1^{(0)}$
$y_2^{(0)}$
$y_K^{(0)}$
$a^{(0)}, \tau^{(0)}$, code
304
308
$\hat{b}_1$
$\hat{b}_2$
$\hat{b}_K$
Re-spread
310
g[t]
312
402
$r_{res}[t]$
Σ
+
-
320
$y_1^{(1)}$
$y_2^{(1)}$
$y_K^{(1)}$
$a^{(1)}, \tau^{(1)}$, code
31.5 GOPS
1,133 GOPS
47.2 GOPS
62.9 GOPS

Figure 4



122
r[t]
112
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
504
$r_{res}[t]$
506
508
312
Re-spread
314
g[t]
$\rho[t]$
$\hat{r}[t]$
r[t]
$a^{(0)}, \tau^{(0)}$, codes
$a^{(1)}, \tau^{(1)}$, codes
31.5 GOPS
8.8 GOPS
0.1 GOP
62.9 GOPS

Figure 5

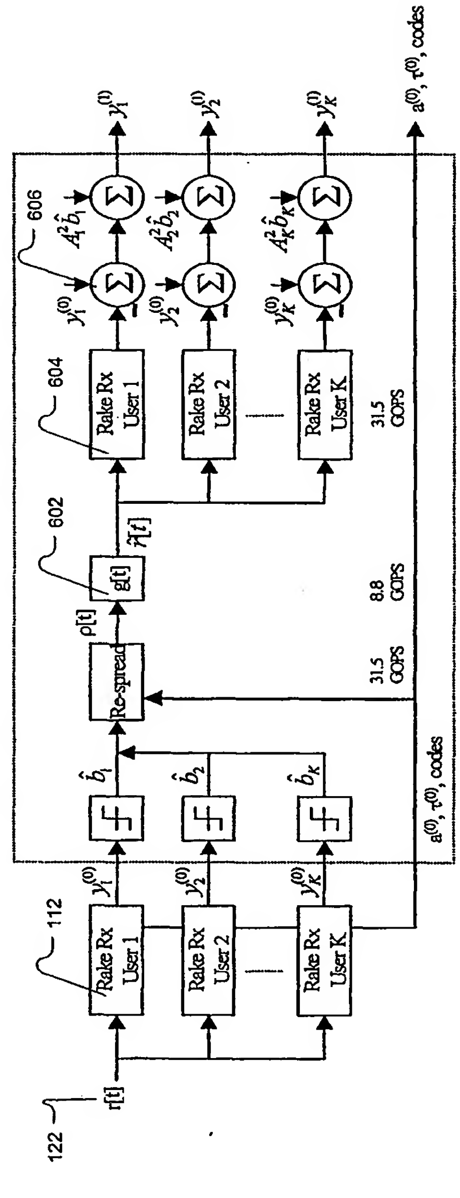

122
r[t]
112
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
$y_1^{(0)}$
$y_2^{(0)}$
$y_K^{(0)}$
$\hat{b}_1$
$\hat{b}_2$
$\hat{b}_K$
Re-spread
$\rho[t]$
g[t]
602
$\hat{r}[t]$
604
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
606
$A_1^2\hat{b}_1$
$A_2^2\hat{b}_2$
$A_K^2\hat{b}_K$
$y_1^{(1)}$
$y_2^{(1)}$
$y_K^{(1)}$
31.5 GOPS
8.8 GOPS
31.5 GOPS
$a^{(0)}, \tau^{(0)}$, codes
$a^{(0)}, \tau^{(0)}$, codes

Figure 6

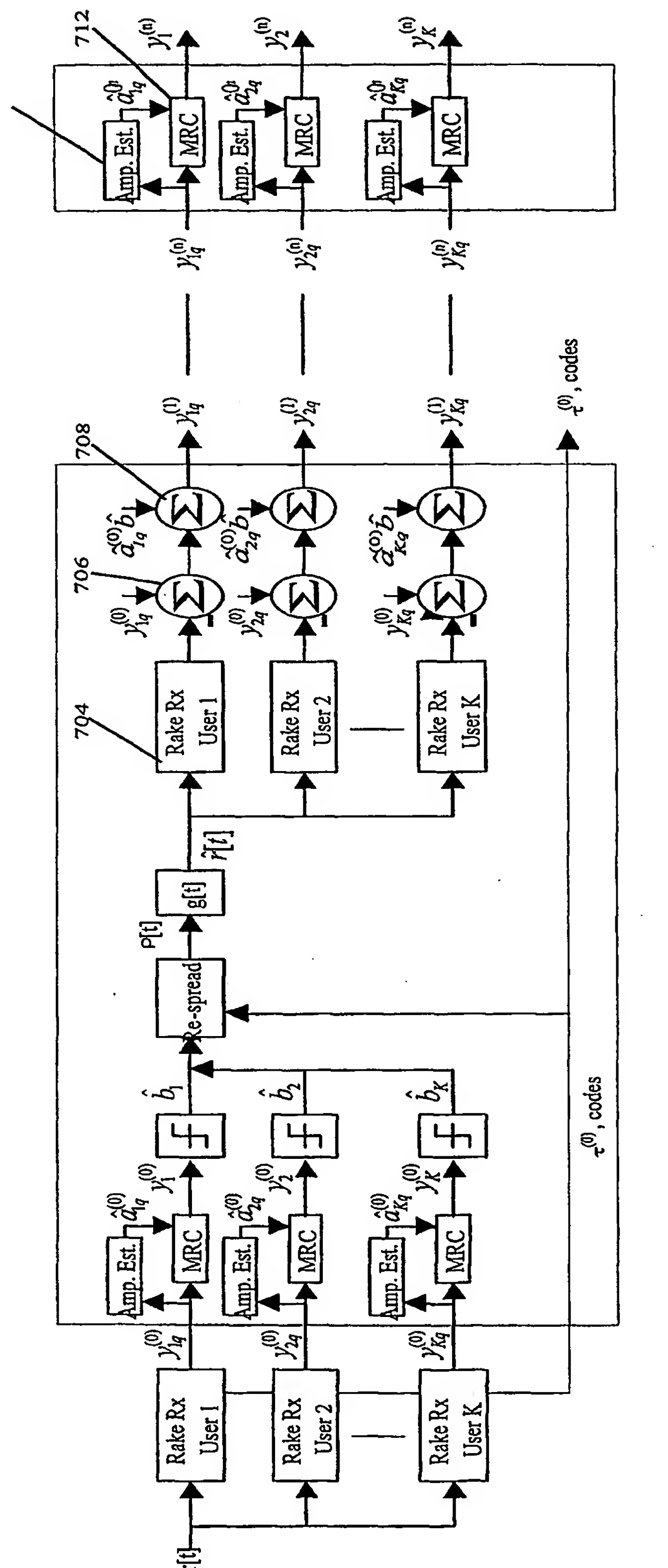

712
708
706
704
Amp. Est.
MRC
Rake Rx User 1
Rake Rx User 2
Rake Rx User K
Re-spread
g[t]
r[t]
$\tau^{(0)}$, codes

Figure 7

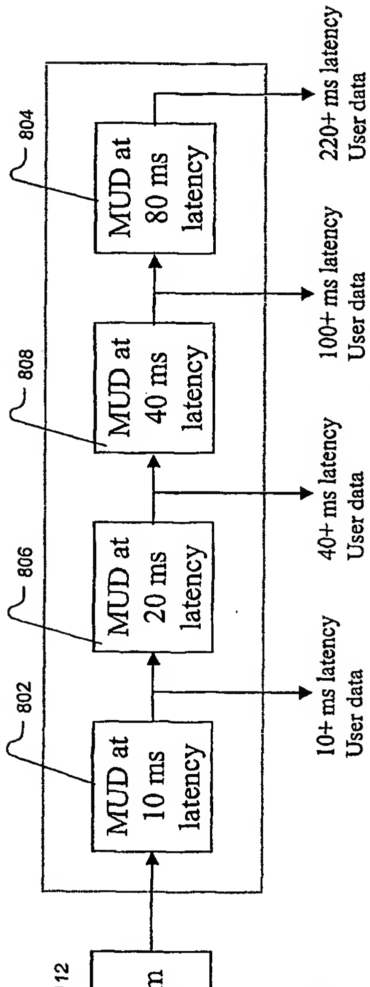


Figure 8

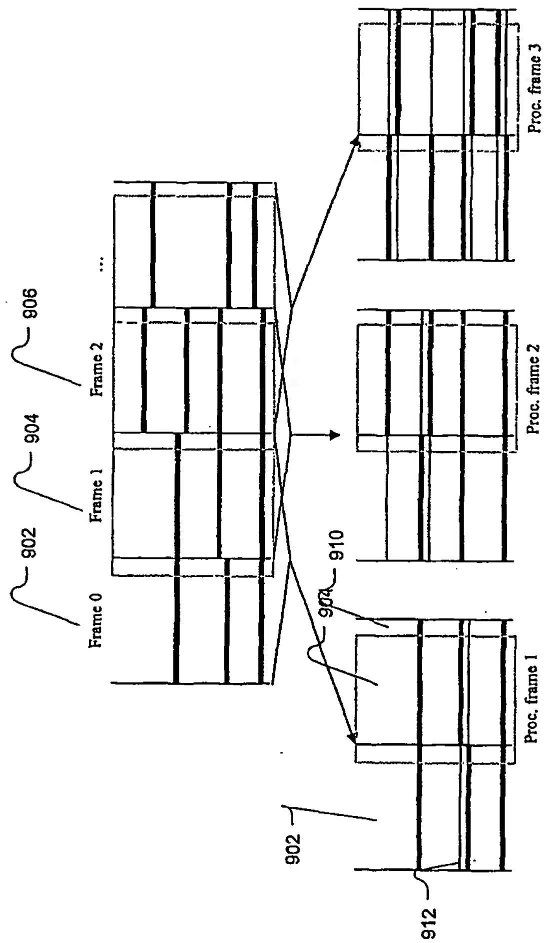


Figure 9

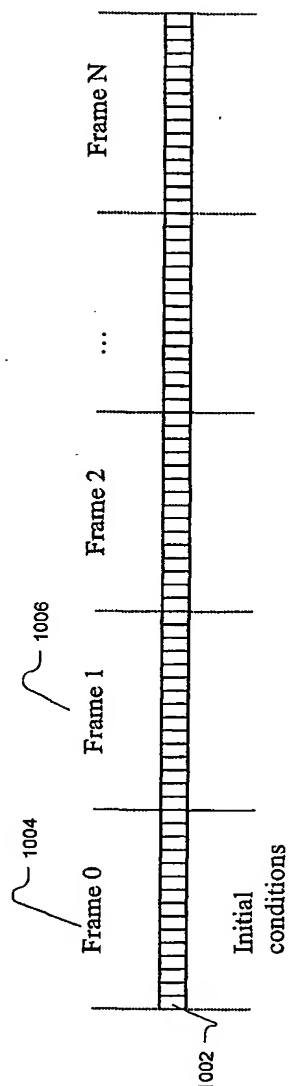

Frame 0
1004
Frame 1
1006
Frame 2
...
Frame N
Initial conditions
1002

Figure 10

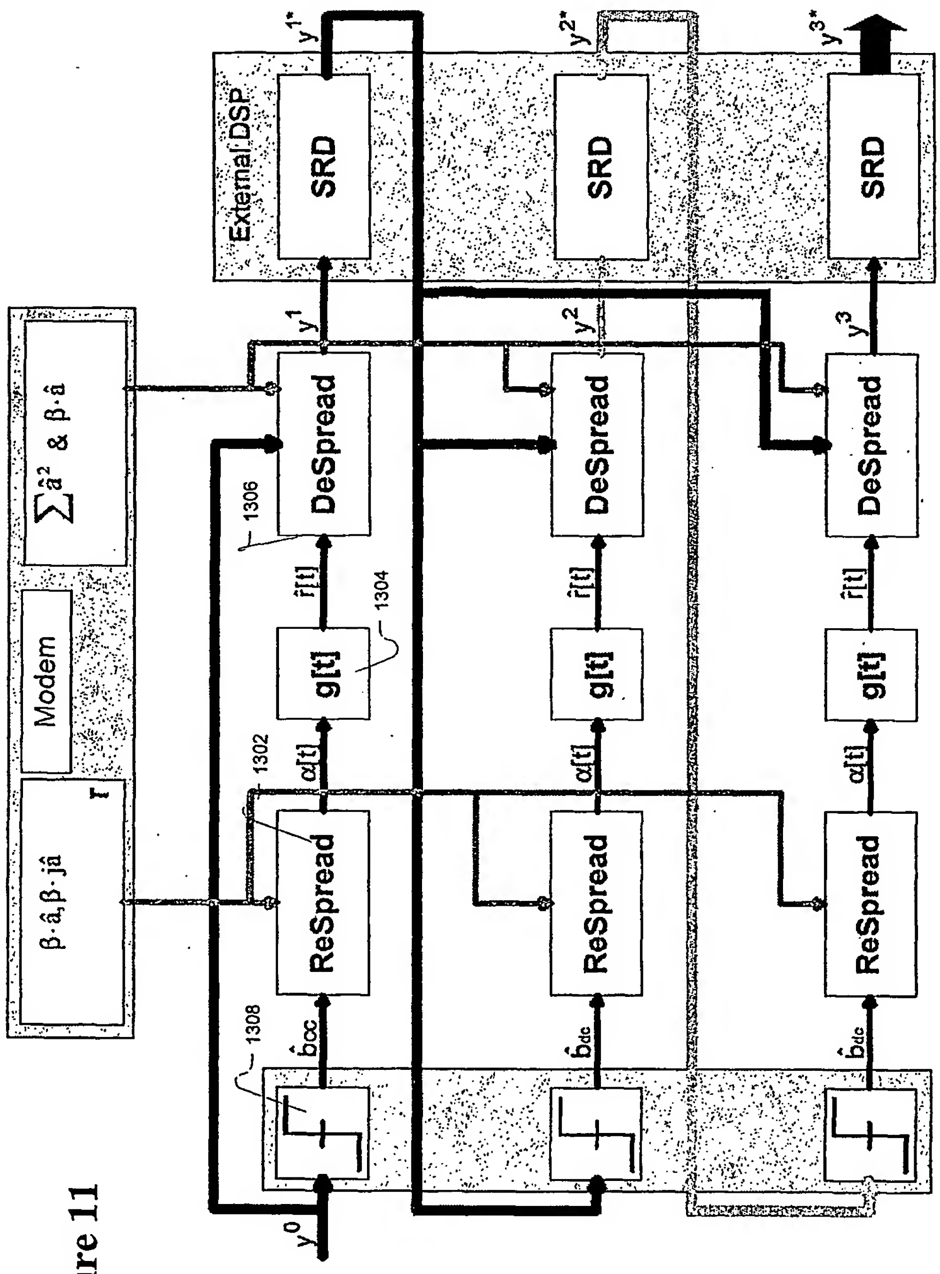

External DSP
SRD
SRD
SRD
y^{1*}
y^{2*}
y^{3*}
DeSpread
DeSpread
DeSpread
y^1
y^2
y^3
$\sum \hat{a}^2$ & $\beta \cdot \hat{a}$
Modem
$\beta \cdot \hat{a}, \beta \cdot j\hat{a}$
r
1306
$\hat{r}[t]$
1304
g[t]
g[t]
g[t]
1302
$\alpha[t]$
ReSpread
ReSpread
ReSpread
1308
$\hat{b}_{cc}$
$\hat{b}_{dc}$
$\hat{b}_{dc}$
y^0

Figure 11

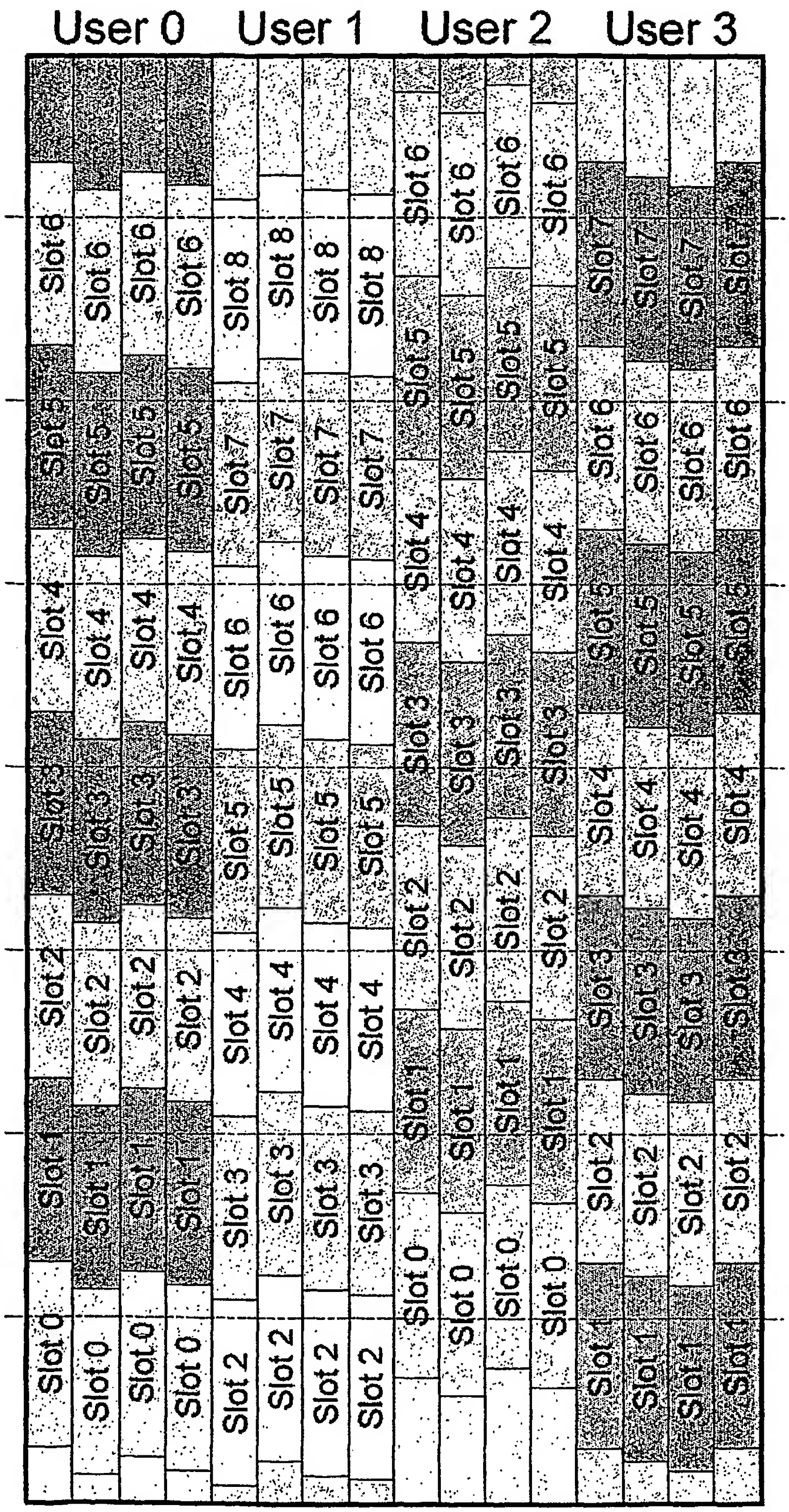

User 0
User 1
User 2
User 3
Slot 0
Slot 1
Slot 2
Slot 3
Slot 4
Slot 5
Slot 6
Slot 7
Slot 8

Figure 12

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: ______________________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.